CS-354

TARUSI MITTAL
1901CS65

END-SEM PART-B

Que 16:-

IMDB ID	Title	Year released	gross (M)	Rating
100	Practical Magic	1998	46	7
101	Dark Knight	2008	532	9
102	Beetle Juice	1988	73	7
103	Heart Breakers	2001	40	6
104	Shrek	2001	267	8
105	The Simpsons Movie	2007	183	8
106	The Holiday	2006	63	7
107	No Time To Die	2021	520	8
108	Mr Queen	2021	340	8
109	Captain Fantastic	2016	280	8
110	Gifted.	2008	102	8
111	The Family	2013	6	7
112	Aladdin	2019	143	7.

We have to find a suitable hashing strategy for the movie records:

→ Hashed File Organisation: Hash file Organisation uses the Computation of Hash function on some fields of the Records. The hash function's output determines the location of block where the records are kept.

We have three options to store this data ie ~~static~~ static hashing, linear hashing or extendible hashing.

For our purpose we are going to use the extendible hashing and so we will see the reasons why we discarded the first two methods.

At first we cannot use static hashing because in it the database size has to be fixed which is not the case for us. In it if the initial no. of buckets is very small, then the performance will degrade due to overflow and if the number becomes too large memory allocated will be very large. Therefor we cannot use static hashing and we have to choose a method from dynamic hashing.

In Dynamic Hashing we have two options either to use linear hashing or go with extendible hashing. We will be using extendible hashing because in linear hashing -

Firstly it does not use a bucket directory.

When an overflow occurs, the pointer bucket splits which means that the bucket which has overflown may or may not be the one to split. Which means that a lot of overflow buckets are created.

So, if we see in the actual world, the no of movies are very high, which means that if we would go with linear hashing then there are a lot of chances that our buckets no is way to high.

So, for our case we are using Extendible Hashing

Extendible Hashing

Extendible Hashing uses hash functions, directories and buckets to hash data and store the records in a random yet uniform way given that we used a good hash function.

For our case we have taken the hashing of movie title to allot random keys to records.

HASH FUNCTION

For our usecase, we will use Polynomial Rolling Hash function

The polynomial hash function is defined as :-

Lets say that the movie title is 's'

$$Hash(s) = [s(0) + s(1) \cdot p + s(2) \cdot p^2 + \dots + s(n-1) \cdot p^{n-1}] \bmod m$$

where, m, p are both positive numbers.

p = It is generally a prime no and we choose it approximately equal to the number of input characters.

So, our now we are assuming that $p = 29$, as mostly movie names have less than 31 characters.

m = m over here is a very large number since the probability of two random strings colliding is $\approx \frac{1}{m}$

So we are assuming m to be $10^9 + 9$

Generating Hash Values

Generally we do the hashing by hand, but in our case the records are quik complex and they require a lot of calculation therefore we have used a C++ code to generate our hash values for the given data.

In our function

$$hash(s) = (s(0) + s(1) \cdot p + s(2) \cdot p^2 + \dots + s(n-1) \cdot p^{n-1}] \bmod m$$

$s(i)$ denotes the ASCII value of the i^{th} character of string.

eg ASCII value of

0 → 48	a → 97
1 → 49	b → 98
2 → 50	c → 99

The C++ code for our usecase is given below:

main.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;

long long findHashValue(string movie) {
    const int m = 1e9 + 9;
    const int p = 29;
    long long ans = 0;
    long long powerUp = 1;
    for (char c : movie) {
        ans = (ans + (int)(c) * powerUp);
        ans = ans % m;
        powerUp = powerUp * p;
        powerUp = powerUp % m;
    }
    return ans;
}

void convertBinary(long long int number){
    int representation[32];

    int count = 0;
    while (number > 0){
        representation[count] = number % 2;
        number = number / 2;
        count++;
    }
    for (int i = count - 1; i >= 0; i--)
        cout << representation[i];
}

int main(){
    int testCase;
    cin>>testCase;
    cin>>ws;
    for(int i=0;i<testCase;i++){
        string s;
        getline(std::cin, s);

        long long int x = findHashValue(s);
        cout<<s<<endl;
        cout<<x<<endl;
        convertBinary(x);
        cout<<endl<<endl;
    }

    return 0;
}
```

Compiled Successfully. memory: 3368 time: 0 exit code: 0

```
Practical Magic
238009119
11100010111110111101100011111

Dark Knight
402826310
11000000000101010010001000110

Beetle Juice
787552802
10111011110001000110100010010

Heart Breakers
221877205
11010011100110010011111010101

Shrek
78241329
10010101001110111100011001

The Simpsons Movie
979252036
11101001011110001100110100100

The Holiday
365459057
10101110010000111011001110001

No Time to Die
42466832
10100001111111111000010000

Mr. Queen
877493572
11010001001101011111010100100

Captain Fantastic
730724320
10101110001101111101111100000

Gifted
159814386
10011000011010010010111110010

The Family
716535315
10101010110101011101100010011
```

Computed Hash Values

Movie Title is hashed for our case and we have find the value corresponding to that. Now, as the binary representation is quite large and we here have to work for small no. of records. therefore we are using the last 5 binary digits from the representation.

ID	Title	Hashed No	Binary Representation
100*	Practical Magic	238009119	11111
101*	Dark Knight	402826310	00110
102*	Beetle Juice	787552802	00010
103*	Heart Breakers	221877205	10101
104*	Shrek	78241329	~~10001~~ 10001
105*	The Simpsons Movie	979252036	00100
106*	The Holiday	365459057	10001
107*	No Time To Die	42466832	10000
108*	Mrs Queen	877493572	00100
109*	Captain Fantastic	730724320	00000
110*	Gifted	159814386	10010
111*	The Family	716535315	10011
112*	Aladdin	65 7487126	10110

SCHEMATIC OF THE FILE ORGANIZATION:

Now as the entries are too big, I will be using the IMDB ID to represent one full entry.

$(ID_i)^* := (IMDB\ ID)_i, (Title)_i, (Year\ Released)_i, (Gross(M))_i, (Rating)_i$

Terms used

1. **Buckets** → They are used to store the hashed keys

2. **Directories** → These are the containers that store pointers to bucket.
   No. of directories = $2^{\wedge}$(global depth)

3. **Global Depth** → It is associated with the directories. They denote the number of bits which are used by the hash function to categorize the keys.
   Global Depth = No. of bits in directory.

4. **Local Depth** → It is same as that of Global Depth except for the fact that local depth is associated with the buckets and not the directories.

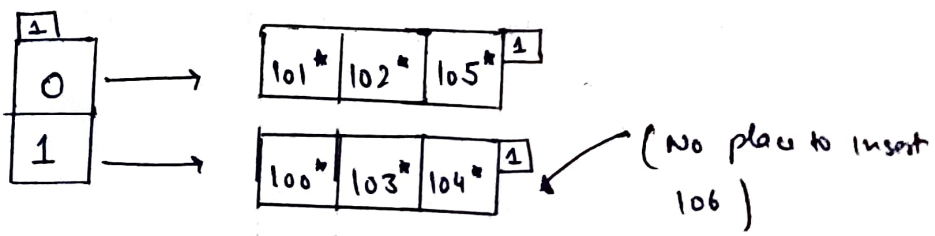
5. **Bucket Splitting** → when the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.

6. **Directory Expansion** → Directory Expansion takes place when a bucket overflows. It is performed when the local depth of the overflowing bucket is equal to the global depth.

INSERTING DATA

We will be inserting data in the order of IMDB Rating.
And we are taking the bucket capacity to be 3

1. **Global Depth = 1**

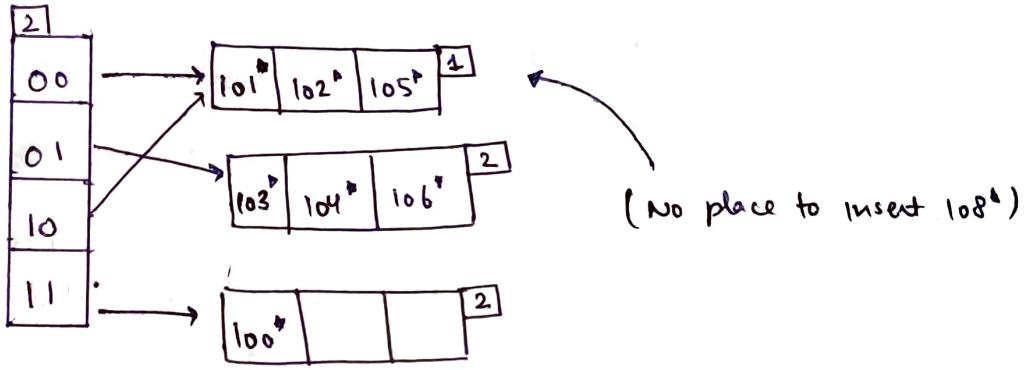


Here when we are trying to insert 106 the 0 bucket is already full so we reached an overflow.

As
Local depth = Global Depth.

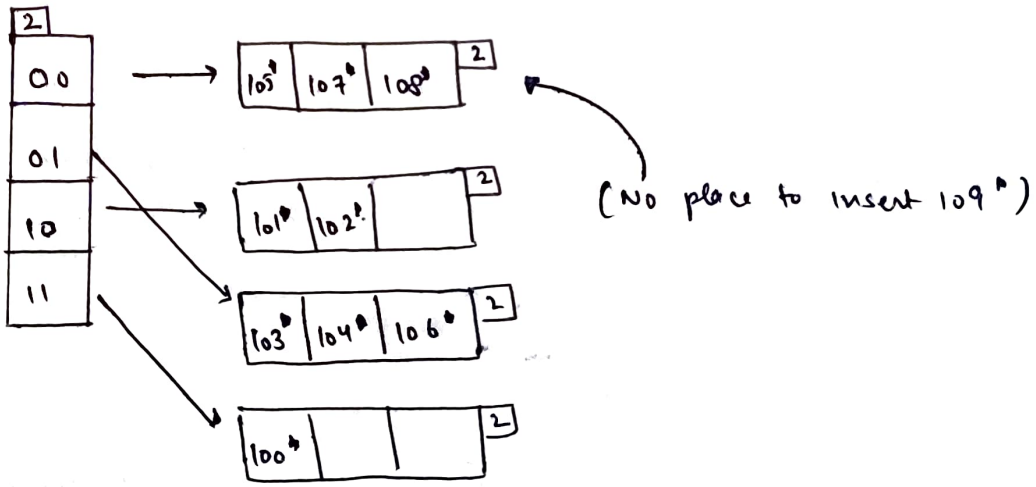
Directory expansion + Splitting will happen.

Global Depth = 2



As local depth < global depth; only bucket splitting takes place.

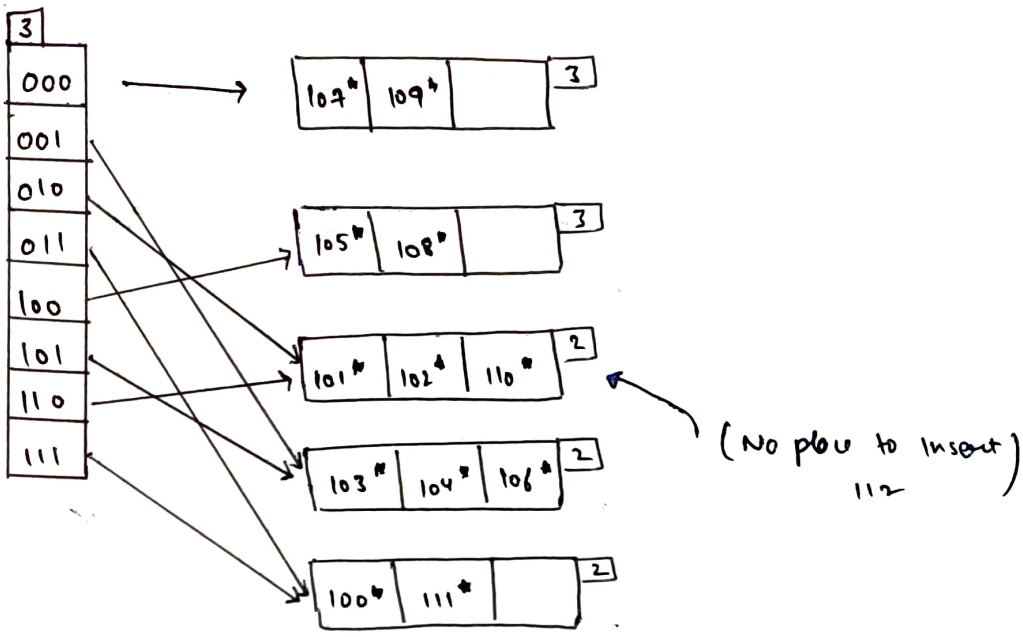
Global Depth = 2



Local depth = global depth.

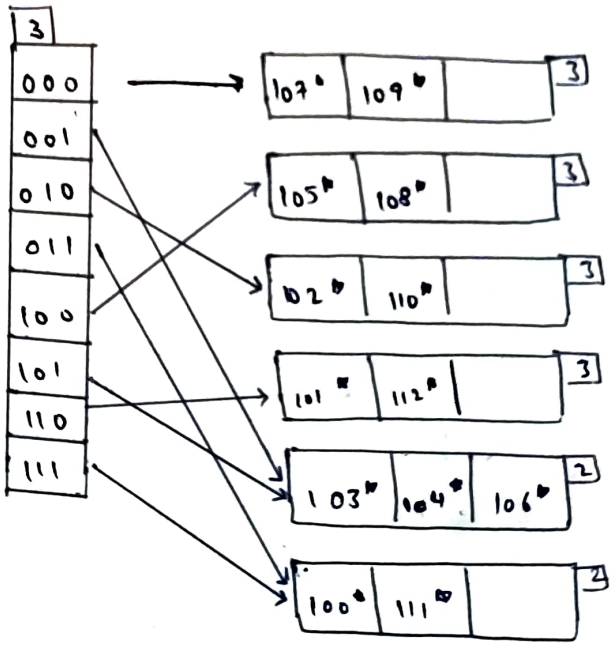
Directory expansion + Splitting takes place.

Global depth = 3



Local depth < Global depth ; only bucket splitting.

Global Depth = 3



This represent our final schematic diagram of file organisation for the given data.

Inserting 5 new Records

Names of Movies Chosen → Hera Pheri, Golmal, Ek Tha Tiger, Tanu Weds Manu, Kabir Singh.

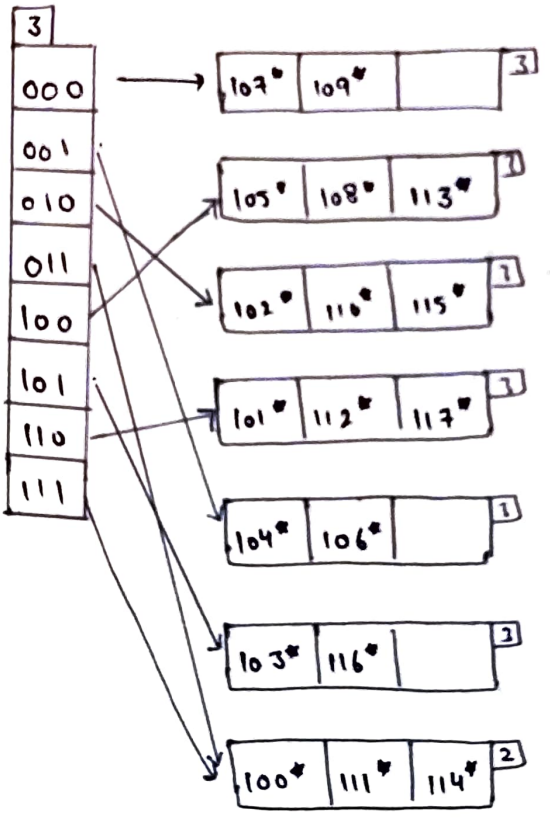
Compiled Successfully. memory: 3436 time: 0.24 exit code: 0

```
Hera Pheri
169427156
10100001100101000000011010100

Golmaal
301858303
10001111111011111110111111111

Ek tha Tiger
938534474
11011111110000111001100100101O

Tanu Weds Manu
213311373
11001011011011011111110001101

Kabir Singh
343400078
10100011101111101111010001110
```

(upto 5)

IMDB ID	Title	Year Released	Gross(in $)	Rating	Hashed No.	Binary Representation
113	Hera Pheri	2004	500	9	169427156	10100
114	Golmal	2001	415	8	301858303	11111
115	Ek tha Tiger	2012	312	9	938534474	01010
116	Tanu weds Manu	2011	240	7	213311373	01101
117	Kabir Singh	2090	300	8	343400078	01110

Now when we will make the updated Schema. at entry 116 the bucket will split and as local depth < global depth only bucket splitting takes place.



So after 5 data entries our hash function generated some new values and the updated schematic is shown above.