DBMS LAB 3

1) a.

The author_id and author_name can be same for many tuple (same author can write many books) , so they can't be candidate keys.The potential candidate key is book id as 2 books can have same name (book name is never copyrighted).

However , here the Book_id has been encoded as **Wo015_Wod** where Wo015 is for author and Wod is the first three letters of the book's name. In that case if the books have different name except for first three characters then potential candidate key is book id + book's name. (Have not considered a case where book name is same because why will an author have 2 books with same name ?).

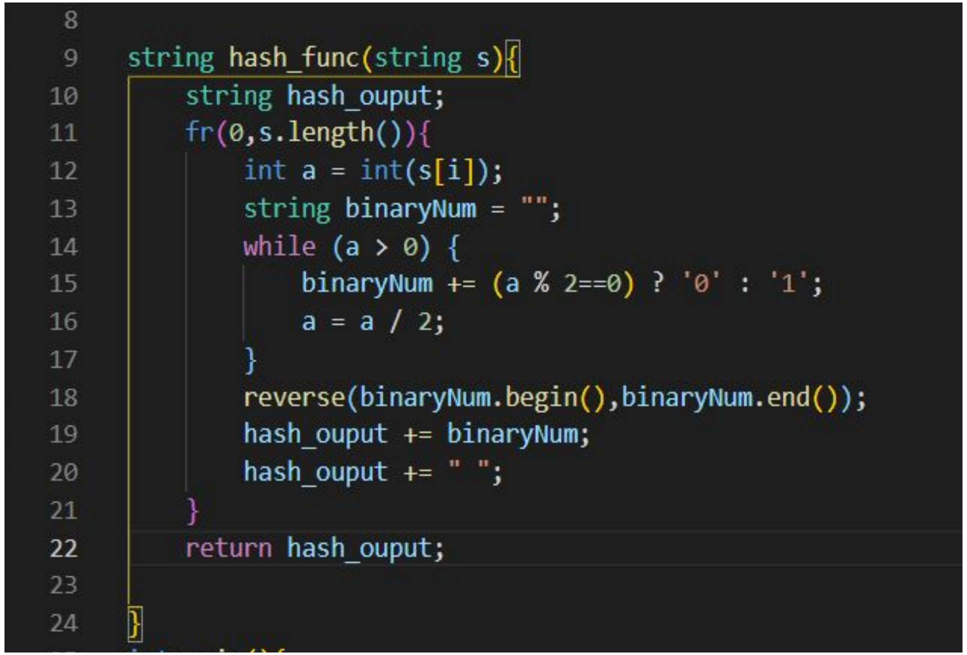
I will use book id + book's name as PK (reason stated above)

However taking 2 domains as search key for hashing is quite difficult . So the book_id can be modified a little bit while encoding like **Wo015_Wod01** , **Wo015_Wod02** (For 2 books with same first three characters and same author).Then this can be used as PK.(Assuming an author will have maximum 99 books with same first 3 characters)

b)

So 1st step is to get the search key.Here I have taken book_id as the search key.

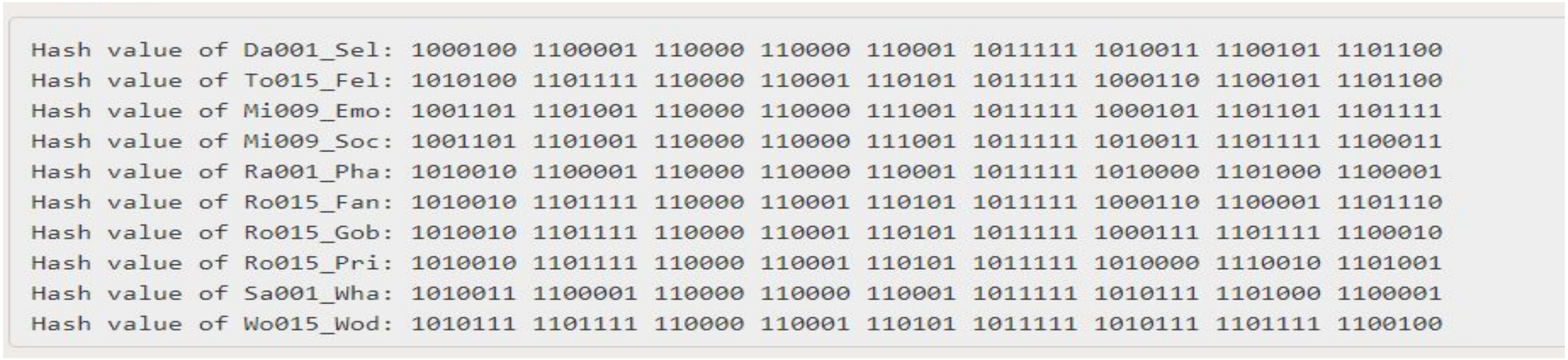
The hash function will convert every char of book_id in the binary form.First it will take the characters then their decimal form from ASCII value , then convert.



```

string hash_func(string s){
    string hash_ouput;
    fr(0,s.length()){
        int a = int(s[i]);
        string binaryNum = "";
        while (a > 0) {
            binaryNum += (a % 2==0) ? '0' : '1';
            a = a / 2;
        }
        reverse(binaryNum.begin(),binaryNum.end());
        hash_ouput += binaryNum;
        hash_ouput += " ";
    }
    return hash_ouput;

}
```

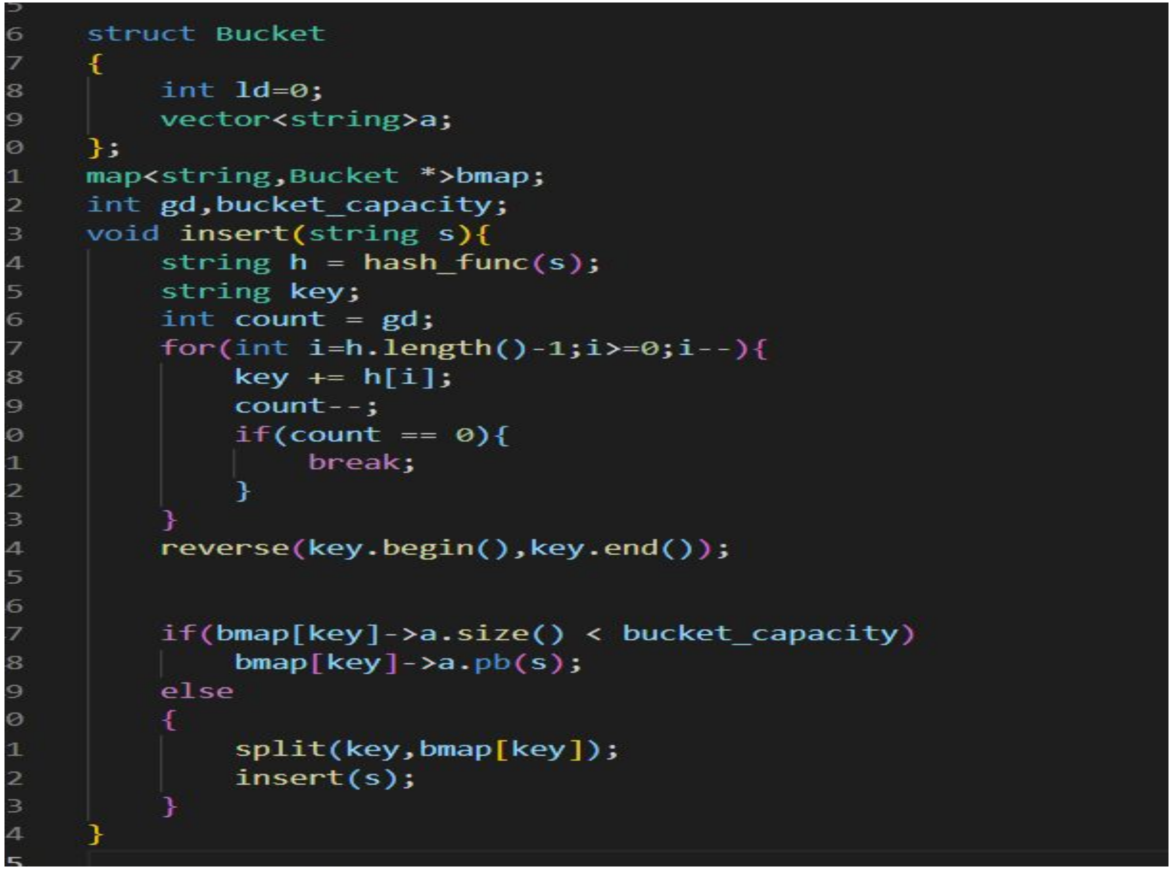


```
Hash value of Da001_Sel: 1000100 1100001 110000 110000 110001 1011111 1010011 1100101 1101100
Hash value of To015_Fel: 1010100 1101111 110000 110001 110101 1011111 1000110 1100101 1101100
Hash value of Mi009_Emo: 1001101 1101001 110000 110000 111001 1011111 1000101 1101101 1101111
Hash value of Mi009_Soc: 1001101 1101001 110000 110000 111001 1011111 1010011 1101111 1100011
Hash value of Ra001_Pha: 1010010 1100001 110000 110000 110001 1011111 1010000 1101000 1100001
Hash value of Ro015_Fan: 1010010 1101111 110000 110001 110101 1011111 1000110 1100001 1101110
Hash value of Ro015_Gob: 1010010 1101111 110000 110001 110101 1011111 1000111 1101111 1100010
Hash value of Ro015_Pri: 1010010 1101111 110000 110001 110101 1011111 1010000 1110010 1101001
Hash value of Sa001_Wha: 1010011 1100001 110000 110000 110001 1011111 1010111 1101000 1100001
Hash value of Wo015_Wod: 1010111 1101111 110000 110001 110101 1011111 1010111 1101111 1100100
```

For implementing extendible hashing , initially assumed that global depth = 1 and local depth = 1.

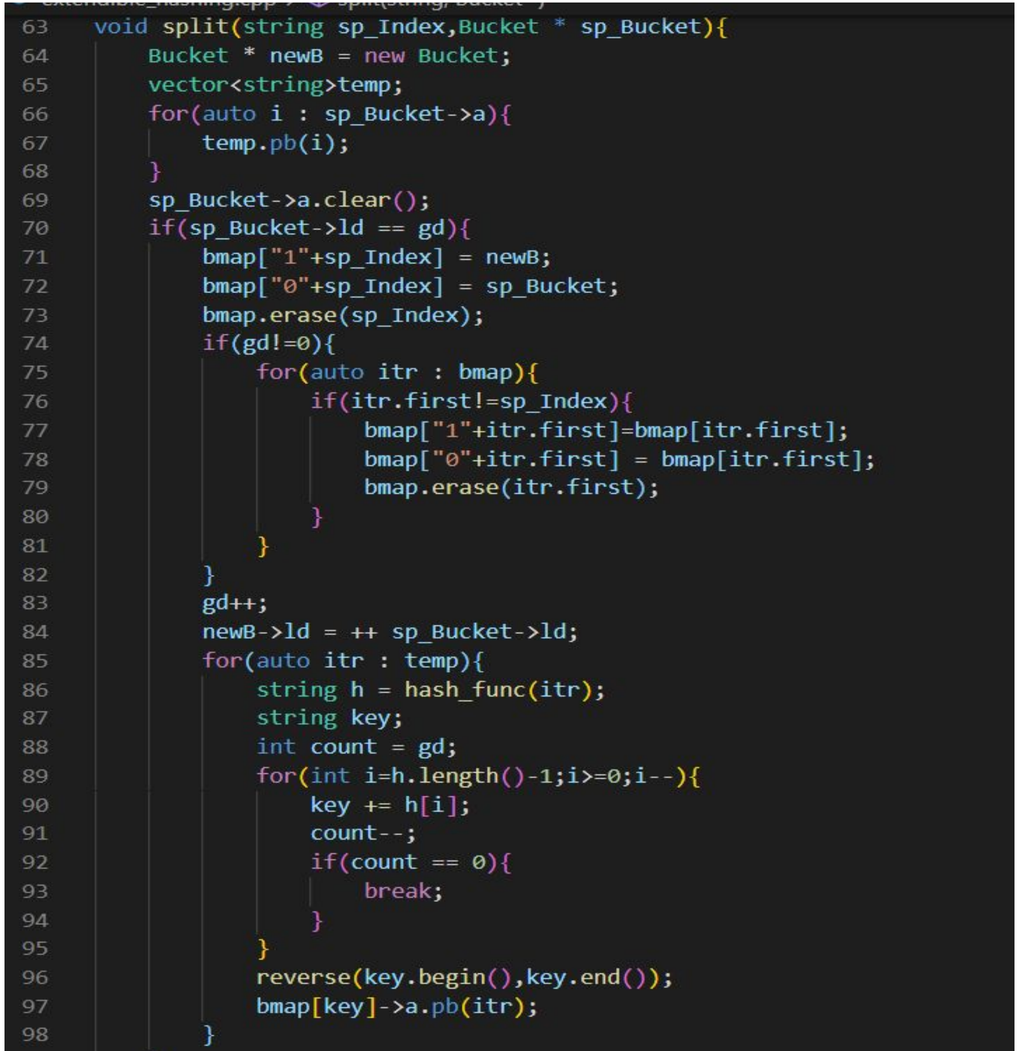
That is LSB is the ans to hash key.

Directory (Global depth = 1)	**Bucket , bucket size = 4 (Local Depth =1)**
0	Da001_Sel , To015_Fel ,Ro015_Fan , Ro015_Gob
1	Mi009_Emo , Mi009_Soc,Ra001_Pha,Ro015_Pri,**Sa001_Wha (OVERFLOW)**

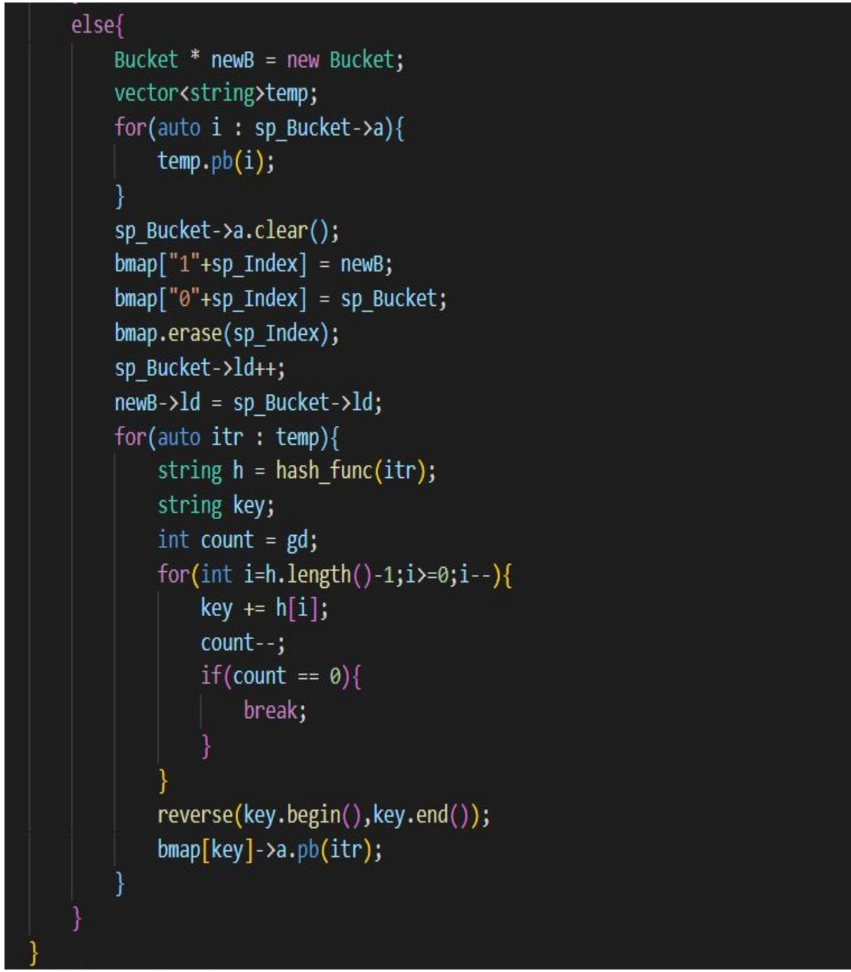


```
struct Bucket
{
    int ld=0;
    vector<string>a;
};
map<string,Bucket *>bmap;
int gd,bucket_capacity;
void insert(string s){
    string h = hash_func(s);
    string key;
    int count = gd;
    for(int i=h.length()-1;i>=0;i--){
        key += h[i];
        count--;
        if(count == 0){
            break;
        }
    }
    reverse(key.begin(),key.end());


    if(bmap[key]->a.size() < bucket_capacity)
        bmap[key]->a.pb(s);
    else
    {
        split(key,bmap[key]);
        insert(s);
    }
}
```



```cpp
void split(string sp_Index,Bucket * sp_Bucket){
    Bucket * newB = new Bucket;
    vector<string>temp;
    for(auto i : sp_Bucket->a){
        temp.pb(i);
    }
    sp_Bucket->a.clear();
    if(sp_Bucket->ld == gd){
        bmap["1"+sp_Index] = newB;
        bmap["0"+sp_Index] = sp_Bucket;
        bmap.erase(sp_Index);
        if(gd!=0){
            for(auto itr : bmap){
                if(itr.first!=sp_Index){
                    bmap["1"+itr.first]=bmap[itr.first];
                    bmap["0"+itr.first] = bmap[itr.first];
                    bmap.erase(itr.first);
                }
            }
        }
        gd++;
        newB->ld = ++ sp_Bucket->ld;
        for(auto itr : temp){
            string h = hash_func(itr);
            string key;
            int count = gd;
            for(int i=h.length()-1;i>=0;i--){
                key += h[i];
                count--;
                if(count == 0){
                    break;
                }
            }
            reverse(key.begin(),key.end());
            bmap[key]->a.pb(itr);
        }
```



```cpp
    else{
        Bucket * newB = new Bucket;
        vector<string>temp;
        for(auto i : sp_Bucket->a){
            temp.pb(i);
        }
        sp_Bucket->a.clear();
        bmap["1"+sp_Index] = newB;
        bmap["0"+sp_Index] = sp_Bucket;
        bmap.erase(sp_Index);
        sp_Bucket->ld++;
        newB->ld = sp_Bucket->ld;
        for(auto itr : temp){
            string h = hash_func(itr);
            string key;
            int count = gd;
            for(int i=h.length()-1;i>=0;i--){
                key += h[i];
                count--;
                if(count == 0){
                    break;
                }
            }
            reverse(key.begin(),key.end());
            bmap[key]->a.pb(itr);
        }
    }
}
```

So when overflow condition occurs and local depth == global depth.The bucket is split and all its data rehashed .The global depth is increased by 1.

Now 2 bits from LSB will be used.

Directory (Global depth = 2)	**Buckets , bucket size = 4**
00 and 10	Da001_Sel , To015_Fel ,Ro015_Fan , Ro015_Gob , Wo015_Wod ()(**Over flow bucket**) (Local depth = 1)
01	Ra001_Pha,Ro015_Pri,**Sa001_Wha** (Local depth = 2)
11	Mi009_Emo, Mi009_Soc (Local depth = 2)

the **local depth of bucket < Global depth** , directories are not doubled but, only the bucket is split and elements are rehashed

Directory (Global depth = 2)	**Buckets , bucket size = 4 (local depth = 2)**
00	Da001_Sel , To015_Fel ,**Wo015_Wod**
10	Ro015_Fan , Ro015_Gob
01	Ra001_Pha,Ro015_Pri,Sa001_Wha
11	Mi009_Emo, Mi009_Soc

c)

For smaller bucket size , taking **bucket size = 2.**

Directory (Global depth = 1)	**Bucket , bucket size = 2 (Local Depth =1)**
0	Da001_Sel , To015_Fel
1	Mi009_Emo , Mi009_Soc,**Ra001_Pha (Overflow)**

Directory (Global depth = 2)	**Buckets , bucket size = 4**
00 and 10	Da001_Sel , To015_Fel ,Ro015_Fan (**Overflow bucket**) (Local depth = 1)
01	Ra001_Pha
11	Mi009_Emo, Mi009_Soc (Local depth = 2)

Directory (Global depth = 2)	**Buckets , bucket size = 4 (local depth = 2)**
00	Da001_Sel , To015_Fel
10	Ro015_Fan , Ro015_Gob
01	Ra001_Pha,Ro015_Pri,**Sa001_Wha(Overflow)**
11	Mi009_Emo, Mi009_Soc

Observation:

So now the directory needs to be further increased to 8 from 4.

So more no. of directory will point to same bucket.It will lead to disadvantages of extendible hashing like Memory being wasted in pointers when the global depth and local depth difference becomes drastic.

So I don’t think keeping bucket size very small is good practice as memory is wasted and splitting the bucket every time can be complicated and time consuming.

The collisions are very frequent.

Directory (Global depth = 1)	Bucket , bucket size = 2 (Local Depth =1)
0	Da001_Sel , To015_Fel ,Ro015_Fan , Ro015_Gob
1	Mi009_Emo , Mi009_Soc,Ra001_Pha,Ro015_Pri,Sa001_Wha

Now, For higher bucket size like 8 or 10.

Observation

There are only 10 records in the table right now , so it got inserted without collision.

I think having larger bucket size for big databases with higher no. records is more effective as it reduces collision hence splitting.However larger bucket size also increase the time complexity insertion,searching and delete.

However if the database is small (smaller than bucket size also) then it will lead to a lot of memory wastage.So then bucket size should be chosen accordingly.

In this case as a library database is usually bigger we can have more bucket size .

d)

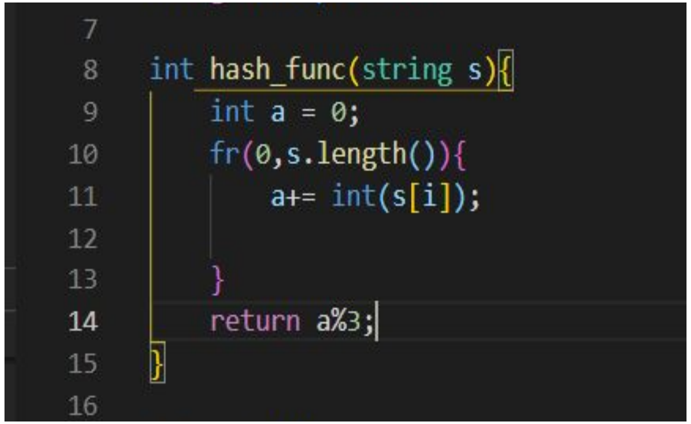
For linear Hashing :

I have used hash_function where

PK (Book_id) is converted into decimal using ASCII .

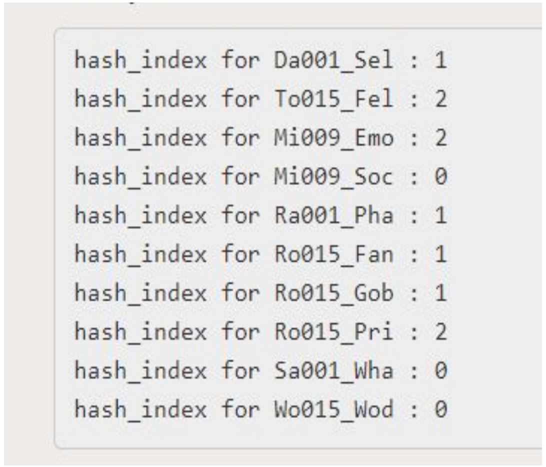
All the decimal values are added and then %3 (As there are 10

records and bucket size is 4).



```
int hash_func(string s){
    int a = 0;
    fr(0,s.length()){
        a+= int(s[i]);

    }
    return a%3;
}
```



```
hash_index for Da001_Sel : 1
hash_index for To015_Fel : 2
hash_index for Mi009_Emo : 2
hash_index for Mi009_Soc : 0
hash_index for Ra001_Pha : 1
hash_index for Ro015_Fan : 1
hash_index for Ro015_Gob : 1
hash_index for Ro015_Pri : 2
hash_index for Sa001_Wha : 0
hash_index for Wo015_Wod : 0
```

Index	Buckets (size = 4)
0 (s==0) if overflow occurs this bucket will split first.	Mi009_Soc , Sa001_Wha , Wo015_Wod
1	Da001_Sel , Ra001_Pha,Ro015_Fan,Ro015_Gob
2	To015_Fel , Mi009_Emo,Ra015_Pri

Now when new entry comes at index 1 there will be overflow. If there is a overflow in case of linear hashing . The first bucket splits firsts into s and M+1 (M is the index of last bucket present) and then s increases. The overflow data is stored in overflow bucket.

e)

If we have larger bucket size such as 15 or 20.We actually just need one bucket to store given datas (10 in number).So we can start with first hash_fucntion as kay%1 which will give 0.

This will decrease the number of collisions .However it will cause a lot of memory wastage .

The time complexity to insert into the bucket will get worse (It is o(bucket size) as bucket size increase the time complexity increase).

So here for small data it is leading to time and memory wastage.

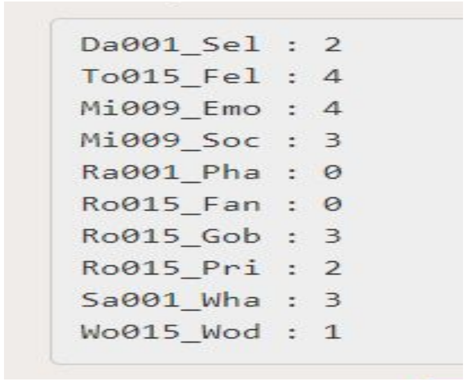
Hence small bucket size of 4 is more effective in this case in terms of time complexity as well as memory wastage.

Further if we reduce the bucket size to 2 . The hash function will be **key % 5**.

As we can see in output 3 is repeated 3 times , so it causes overflow.

So we can get to a smaller bucket size in order to avoid time complexity and memory

Wastage but making bucket size too small can again lead to collision.



```
Da001_Sel  :  2
To015_Fel  :  4
Mi009_Emo  :  4
Mi009_Soc  :  3
Ra001_Pha  :  0
Ro015_Fan  :  0
Ro015_Gob  :  3
Ro015_Pri  :  2
Sa001_Wha  :  3
Wo015_Wod  :  1
```

f)

a) Sa_001, Sa001_Voy, Safina, Voyage of the Turtle

So hash_function return = `1010011 1100001 110000 110000 110001 1011111 1010110 1101111 1111001`

The last 2 bits are 01.

Directory (Global depth = 2)	**Buckets , bucket size = 4 (local depth = 2)**
00	Da001_Sel , To015_Fel ,Wo015_Wod
10	Ro015_Fan , Ro015_Gob
01	Ra001_Pha,Ro015_Pri,Sa001_Wha, Sa001_Voy
11	Mi009_Emo, Mi009_Soc

b) The hash_func returns index 0.

Index	Buckets (size = 4)
0	Mi009_Soc , Sa001_Wha , Wo015_Wod,Sa001_Voy
1	Da001_Sel , Ra001_Pha,Ro015_Fan,Ro015_Gob
2	To015_Fel , Mi009_Emo,Ra015_Pri

c) In **extendible hashing** first it takes O(8.n)==O(n) time for the conversion of PK into binary .n is equal to the size of Primary key .8.n as every character is converted into 8 bit binary number.Then O(1) i.e. in constant time number of bits for determining the directory from which bucket address needs to be stored is determined. Then inserting in that bucket depends on the number of entries already present in the bucket . In worst case it will be O(bucket size). Here it is the worst case so it is O(bucket size).

In **linear hashing** first it takes O(n) where n is the size of Primary key time to convert the primary key into hash index.Then it takes O(bucket size) time to insert in the bucket.

Obviously the conversion to hash_key is simpler in case of linear hashing as compared to extendible hashing.

Suppose if splitting was to be done then it would have taken further time in splitting and rehashing the elements of bucket (O(bucket size)) .Splitting will take same time in both the cases.However rehashing would take more time in extendible hashing.

For subsequent retrieval of record also same time complexity analysis will come as same as for inserting.

g)

<Ro_015, Ro015_Phi, Rowling, Philosopher’s Stone_Harry Potter>

a) Hash function will generate 1010010 1101111 110000 110001 110101 1011111 1010000 1101000 1101001

And last 2 bit 01.

Directory (Global depth = 2)	**Buckets , bucket size = 4 (local depth = 2)**
00	Da001_Sel , To015_Fel ,Wo015_Wod
10	Ro015_Fan , Ro015_Gob
01	Ra001_Pha,Ro015_Pri,Sa001_Wha, Sa001_Voy , **Ro015_Phi (Overflow)**
11	Mi009_Emo, Mi009_Soc

As the global depth == local depth . No. of directories are multiplied by 2 and overflowing bucket is split into 2.

All the elements of 01 are rehashed taking into consideration 3 bits. However still athe elements are in the same bucket.

Directory (Global depth = 3)	**Buckets , bucket size = 4**
000 , 100 (Both directory point to same bucket)	Da001_Sel , To015_Fel ,Wo015_Wod (Local depth = 2)
010 , 110(Both directory point to same bucket)	Ro015_Fan , Ro015_Gob (Local depth = 2)
001	Ra001_Pha,Ro015_Pri,Sa001_Wha , Sa001_Voy **, Ro015_Phi (Overflow) (LD = 3)**
101	LD=3
111,011	Mi009_Emo, Mi009_Soc (LD = 2)

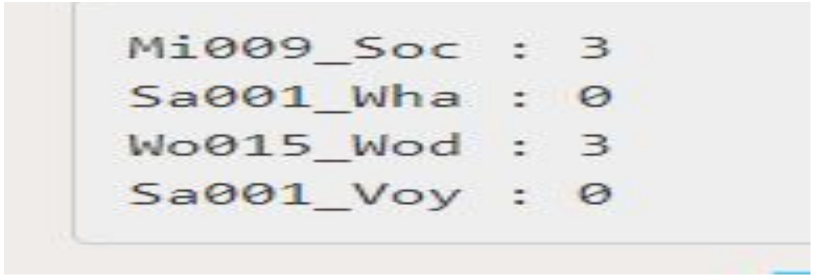
Now as LD == GD again directory should be multiplied by 2 and overflowing bucket is split into 2.

Directory (Global depth = 3)	Buckets , bucket size = 4
0000 , 0100 , 1000,1100 (All the directory point to same bucket)	Da001_Sel , To015_Fel ,Wo015_Wod (LD = 2)
0010 , 0110,1010 , 1110(All the directory point to same bucket)	Ro015_Fan , Ro015_Gob (LD = 2)
0001	Ra001_Pha,Sa001_Wha (Ld = 4)
1001	Ro015_Pri , Sa001_Voy, Ro015_Phi (LD = 4)
0101,1101(All the directory point to same bucket)	LD = 4
0111,0011,1111,1011(All the directory point to same bucket)	Mi009_Emo, Mi009_Soc (LD = 2)

b) So for linear hashing hash function will give hash key as 1.

Index	Buckets (size = 4)
0 (s==0)	Mi009_Soc , Sa001_Wha , Wo015_Wod,Sa001_Voy
1	Da001_Sel , Ra001_Pha,Ro015_Fan,Ro015_Gob , **Ro015_Phi (Overflow)**
2	To015_Fel , Mi009_Emo,Ra015_Pri

So s == 0 i.e first bucket 0 will be split into 0 and new bucket 3 and pointer (s) will be incremented to 1.So the elements of bucket 0 will split between 0 and 3 using another hash function **key % 6.**



```
Mi009_Soc  :  3
Sa001_Wha  :  0
Wo015_Wod  :  3
Sa001_Voy  :  0
```

Index	Buckets (size = 4)
0	Sa001_Wha , Sa001_Voy
1 (s==1)	Da001_Sel , Ra001_Pha,Ro015_Fan,Ro015_Gob , **Ro015_Phi (IN Overflow bucket)**
2	To015_Fel , Mi009_Emo,Ra015_Pri
3	Mi009_Soc , Wo015_Wod,

So when another value is inserted and it cause overflow bucket at 1 will split adjusting the record added in overflow bucket.

c) In **extendible hashing** first it takes O(8.n)==O(n) time for the conversion of PK into binary .n is equal to the size of Primary key .8.n as every character is converted into 8 bit binary number.Then O(1) i.e. in constant time number of bits for determining the directory from which bucket address needs to be stored is determined. Then inserting in that bucket depends on the number of entries already present in the bucket . In worst case it will be O(bucket size). Here it is the worst case so it is O(bucket size).

However this time splitting also occur which is causing a lot of memory wastage in extendible hashing . After the splitting of 01 directory and bucket and rehaing all the element go into 001 and nothing in 101 . As a result another

Splitting needs to happen which is time consuming as well as memory wastage.In such cases where many books of the database have same first three character other hash functions can make extendible hashing better.

In **linear hashing** first it takes O(n) where n is the size of Primary key time to convert the primary key into hash index.Then it takes O(bucket size) time to insert in the bucket.

In linear hashing just one split is happening .Actual in linear hashing no matter what every time whenever there is an overflow just one split happens to the bucket which is being currently pointed and that overflow data is stored in overflow bucket.

ALL the time complexity analysis is valid to searching as well.

Clearly for this case linear hashing is more efficient both time and memory wise because of it one split at a time nature. (But in some cases more and more data can get in overflow bucket which will inturn increase searching and inserting time , in that case extendible hashing will be better).

h)

Yes , I have better way to encode author_id and book_id .A book_id is encoded as Ro015_Phi where Ro015 is author id and Phi are the 1st three characters of book’s name.

However what if an author writes 2 books with same three initial characters. In that case book_id will no more be the PK as PK needs to be unique.Solution for this problem is book_id should be encoded as Ro015_Phi_01 , Ro015_Phi_02 like this.

However the hash function we are using for extendible hashing takes in account least significant bit .If we have numbering of books placed on right side it will lead to a lot of collisions (lot of 01 numbered books).

So instead of placing it at the right side it can be placed in the middle. Like this :

Ro015_01_Phi , Ro015_02_Phi and so on.

This will also reduce collision.

I will not put on the extreme left because if want to use author’s name as a search key then it will cause trouble.

Also the author id can be made of first three characters instead of 2 . Like Saf_001. This makes it more unique (if to be used as a search key can avoid a lot of collisions as chances of having first 2 characters as same for a name becomes very high).

This in turn will also change book _id to Saf001_01_Wha.