UNIT-II: HASHING

Introduction - Static Hashing - Hash Table- Hash Functions - Secure Hash Function - Overflow Handling - Theoretical Evaluation of Overflow Techniques, Dynamic Hashing - Motivation for Dynamic Hashing - Dynamic Hashing Using Directories - Directory less Dynamic Hashing.

Introduction:

When a dictionary with n entries is represented as a binary search tree, the dictionary operations search, insert, and delete take **O(n)** time. These dictionary operations may be performed in **O(log n)** time using a balanced binary search tree. In this chapter, we examine a technique called hashing, that enables us to perform the dictionary operations search, insert and delete in **O(1)** time expected.

Hashing:

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities or colleges, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values.

In hashing, large keys are converted into small keys by using **hash functions**. The values are then stored in a data structure called **hash table**. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in **O(1)** time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hashing is implemented in two steps:

1. **An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.**

2. **The element is stored in the hash table where it can be quickly retrieved using hashed key.**

There are many possibilities for representing the dictionary and one of the best methods for representing is hashing. Hashing is a type of a solution which can be used in almost all situations. Hashing is a technique which uses less key comparisons. This method generally used the hash functions to map the keys into a table, which is called a hash table.

Hashing is of two types.

1) Static Hashing and
2) Dynamic Hashing

Static Hashing

1) Hash Table

Hash table is a type of data structure which is used for storing and accessing data very quickly. Insertion of data in a table is based on a key value. Hence every entry in the hash table is defined with some key. By using this key data can be searched in the hash table by few key comparisons and then searching time is dependent upon the size of the hash table.

2) Hash Function

Hash function is a function which is applied on a key by which it produces an integer, which can be used as an address of hash table. Hence one can use the same hash function for accessing the data from the hash table. In this the integer returned by the hash function is called hash key.

Types of Hash Functions

There are various types of hash function which are used to place the data in a hash table,

1) **Division Method**
2) **Mid square Method**
3) **Digit folding Method**
4) **Digit Analysis Method/Binary/Radix Method**

1)Division Method

In this method the hash function is dependent upon the remainder of a division.

For example if the record **52, 68, 99, 84** is to be placed in a hash table and let us take the table size is 10.

Then:

h(key)= key % table_size
h(52) = 52 % 10 = 2
h(68) = 68 % 10 = 8
h(99) = 99 % 10 = 9
h(84) = 84 % 10 = 4

Division Method

0	
1	
2	52
3	
4	84
5	
6	
7	
8	68
9	99

Hash Table

```
//-------------------------------------------------------------------------------------------
//Program to insert elements into a hash table using Division method
//-------------------------------------------------------------------------------------------
#include <stdio.h>
#include <conio.h>
int a[10],size;

int hashfunction(int e)
{
    int key;
    key = e % size;
    return key;
}

void main()
{
    int i, j, element;

    clrscr();
    printf("enter size of hash table ");
    scanf("%d",&size);

    for(i=0; i<size; i++)
    {
        printf("enter element to insert ");
        scanf("%d",&element);

        j=hashfunction(element);
        a[j]=element;
     }

   printf("values in hash Table\n");
   for(i=0; i<size; i++)
        printf("%d\n",a[i]);

    getch();
  }
```

2. Mid Square Method

In this method firstly key is squared and then mid part of the result is taken as the index. For example: consider that if we want to place a record of **3101** and the size of table is 1000.

So, **3101*3101=9616201**

i.e. h (3101) = 162 (middle 3 digit).

k	k*k	Square	index h(k)
15	15 * 15	225	2
22	22 * 22	484	8
16	16 * 16	256	5
29	29 * 29	841	4
31	31 * 31	961	6
5	5 * 5	25	0
3	3 * 3	9	9

Mid Square Method

0	5
1	
2	15
3	
4	29
5	16
6	31
7	
8	22
9	3

Hash Table

If the square value contains even number of digits the index is 0.

If it contains odd value index is middle value.

If the square value is a single digit, the value will be placed in that index only. Value 3 is stored at location 9.

3.Digit Folding Method

In this method, we partition the identifier k into several parts. All parts, except for the last one have the same length. We then add the parts together to obtain the hash address for k. There are two ways of carrying out this addition. In the first method, we shift all parts except for the last one, so that the least significant bit of each part lines up with the corresponding bit of the last part. We then add the parts together to obtain h(k). This method is known as **shift folding.**

The second method, known as **folding at the boundaries**, the key is folded at the partition boundaries, and digits falling into the same position are added together to obtain h(k). This is equivalent to reversing every other partition before adding.

Example 1: Suppose that k=12320324111220, and we partition it into parts that are 3 decimal digits long. The partitions are P1=123, P2=203, P3=241, P4=112 and P5=20

Shift Folding

$$h(k) = \sum_{i=1}^{5} P_i$$

= P1+P2+P3+P4+P5
= 123 + 203 + 241 + 112 + 20
= 699

Folding at the boundaries

When folding at boundaries is used, we first reverse P2 and P4 to obtain 302 and 211 respectively. Next the five partitions are added to obtain

$$h(k) = \sum_{i=1}^{5} P_i$$

= P1+P2+P3+P4+P5
= 123 + 302 + 241 + 211 + 20
= 897

Example 2) For example: consider a record of 12465512 then it will be divided into parts.
i.e. 124, 655, 12. After dividing the parts combine these parts by adding it.

Shift folding
H(key) =124+655+12 = 791
791 is the index to store the value 12465512

Folding at the boundaries
H(key) =124+556+12 = 692
692 is the index to store the value 12465512

4) Digit Analysis Method:

In this method we will examine, digit analysis, is used with static files. A static file is one in which all the identifiers are known in advance.

Using this method, we first transform the identifiers into numbers using some radix, r.

We then examine the digits of each identifier, deleting those digits that have the most skewed distributions. We continue deleting digits until the number of remaining digits is small enough to give an address in the range of the hash table. The digits used to calculate the hash address must be the same for all identifiers and must not have abnormally high peaks or valleys (the standard deviation must be small).

For Example store values 4,8,3,7 in the hash table considering the radix 2.

Consider the hash table having a size of 8 elements. The index is in binary. The no of digits in index is 3. So we are going to generate 3 digited index from our has function.

k	value in binary	h(k)
4	0100	100
8	1000	000
3	0011	011
7	0111	111

Digit Analysis Method

000	8
001	
010	
011	3
100	4
101	
110	
111	7

Hash Table

Characteristics of Good Hashing Function

1) The hash function should generate different hash values for the similar string.

2) The hash function is easy to understand and simple to compute.

3) The hash function should produce the keys which will get distributed, uniformly over an array.

4) A number of collisions should be less while placing the data in the hash table.

5) The hash function is a perfect hash function when it uses all the input data.

Collision

It is a situation in which the hash function returns the same hash key for more than one record, it is called as collision. Sometimes when we are going to resolve the collision it may lead to a overflow condition and this overflow and collision condition makes the poor hash function.

Collision resolution technique

If there is a problem of collision occurs then it can be handled by apply some technique. These techniques are called as collision resolution techniques. There are generally four techniques which are described below.

1) Chaining

2) Linear Probing (Open addressing)

3) Quadratic Probing (Open addressing) and

4) Double Hashing (Open addressing).

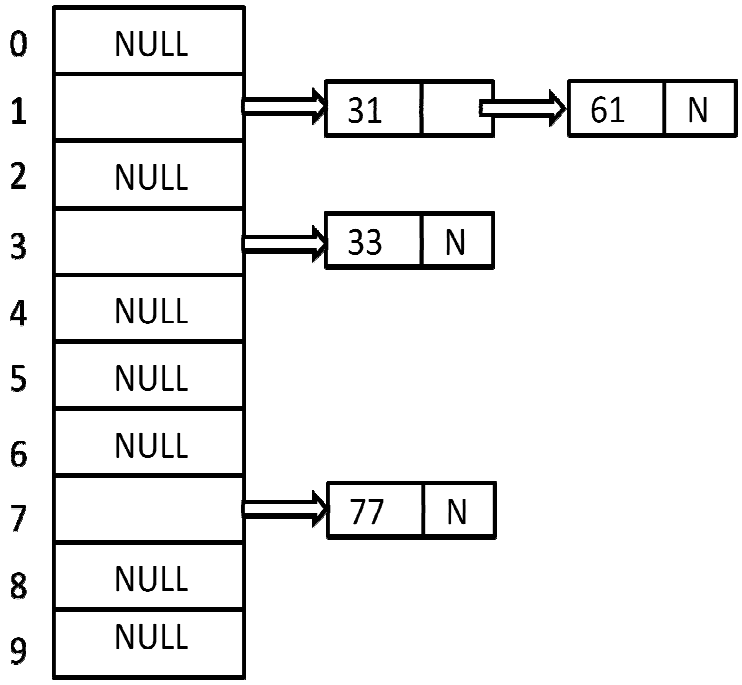
1) Chaining

It is a method in which additional field with data i.e. chain is introduced. A chain is maintained at the home bucket. In this when a collision occurs then a linked list is maintained for colliding data.

For Example: Let us consider a hash table of size 10 and we apply a hash function of H(key)=key % size of table. Let us take the keys to be inserted are **31, 33, 77, 61**.

In the diagram we can see at same bucket 1 there are two records which are maintained by linked list or we can say by chaining method.

H(31) = 31%10 = 1
H(33) = 33%10 = 3
H(77) = 77%10 = 7
H(61) = 61%10 = 1



2) Linear probing (Open addressing)

It is very easy and simple method to resolve or to handle the collision. In this, collision can be solved by placing the second record linearly down, whenever the empty place is found. In this method there is a problem of clustering which means at some place block of a data is formed in a hash table.

Example: Let us consider a hash table of size 10 and hash function is defined as H(key)=key % table_size. Consider that following keys are to be inserted that are **56, 64, 36, 71**.

56 % 10 = 6
64 % 10 = 4

0	NULL
1	NULL
2	NULL
3	NULL
4	64
5	NULL
6	56
7	NULL
8	NULL
9	NULL

36 % 10 = 6

The index 6 is already filled with 56
It is not empty
Collision occurred
To resolve this check the next location
i.e. 6+1 = 7
index 7 is NULL so insert 36 at index 7.

0	NULL
1	NULL
2	NULL
3	NULL
4	64
5	NULL
6	56
7	36
8	NULL
9	NULL

71 % 10 = 1

As the index 1 is null
We can insert 71 at index 1

0	NULL
1	71
2	NULL
3	NULL
4	64
5	NULL
6	56
7	36
8	NULL
9	NULL

In this diagram we can see that 56 and 36 need to be placed at same bucket but by linear probing technique the records linearly placed downward if place is empty i.e. it can be seen 36 is placed at index 7.

```
//-----------------------------------------------------------------------------------------------
//Program to insert elements into a hash table using linear probing - Division method
//-----------------------------------------------------------------------------------------------
#include <stdio.h>
#include <conio.h>
int a[10], size;

int hashfunction(int e)
{
  int key;
  key = e % size;
  if (a[key]==0)
    return key;
  else
    if (size==key)
     {
       printf("hash table is FULL");
       return -1;
     }
    else
      hashfunction(e+1);
}

void main()
{
    int i,j,element;
    clrscr();
    printf("enter size of hash table ");
    scanf("%d",&size);

        // ----------------------------------------------initialise hash table
   for(i=0; i<size; i++)
     a[i]=0;

    for(i=0; i<size; i++)
    {
        printf("enter element to insert ");
        scanf("%d",&element);
        j=hashfunction(element);
        a[j]=element;
     }

    printf("values in hash Table\n");
    for(i=0; i<size; i++)
       printf("%d\n",a[i]);

    getch();
  }
```

3) Quadratic Probing (Open addressing)

This is a method in which solving of clustering problem is done. In this method the hash function is defined by the

H(key)=(H(key)+x*x)%table_size

Let us consider we have to insert following elements that are:-
67, 90, 55, 17, 49.

67%10 = 7

90%10 = 0

55%10 = 5

17%10 = 7

49%10 = 9

0	90
1	
2	
3	
4	
5	55
6	
7	67
8	
9	

In this we can see if we insert 67, 90, and 55 it can be inserted easily but in the case of 17 hash function is used in such a manner that :-

To insert 17

The initial index generated is 17%10 = 7

But the index 7 is already filled with 67. Collision occurred.
To resolve this we try

(7+0*0)%10 = 7

(when x=0 it provide the index value 7 only) by making the increment in value of x. let x =1 so ,

(7+1*1)%10 = 8.

in this case bucket 8 is empty hence we will place 17 at index 8.

67%10 = 7

90%10 = 0

55%10 = 5

17%10 = 7

49%10 = 9

0	90
1	
2	
3	
4	
5	55
6	
7	67
8	17
9	49

4) Double hashing (Open addressing)

It is a technique in which two hash functions are used when there is an occurrence of collision. In this method 1 hash function is simple as same as division method. But for the second hash function there are two important rules which are

1. It must never evaluate to zero.
2. Must sure about the buckets, that they are probed.

The hash functions for this technique are:

H1(key) = key % table_size
H2(key) = P-(key mod P)

Where, **p** is a prime number which should be taken smaller than the size of a hash table.

Example: Let us consider we have to insert **67, 90,55,17,49**.

67%10 = 7
90%10 = 0
55%10 = 5
17%10 = 7
= 7 - (17%7)
= 7 - 3
= 4

49%10 = 9

0	90
1	17
2	
3	
4	
5	55
6	
7	67
8	
9	49

In this we can see 67, 90 and 55 can be inserted in a hash table by using first hash function but in case of 17 the bucket is full and in this case we have to use the second hash function which is

H2(key) = P - (key mod P)

where **P is a prime number which should be taken smaller than the hash table so value of P will be 7.**

i.e. **H2(17) = 7 - (17%7)**
= 7 - 3
= 4

that means we have to take **4 jumps for placing 17**. Therefore 17 will be placed at index 1.

```
/*-------------------------------------------------------
      Program to implement Double Hashing
-------------------------------------------------------*/
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define TABLE_SIZE 10

int h[TABLE_SIZE]={NULL};

void insert()
{

 int key,index,i,hkey,hash2;

 printf("\nenter a value to insert into hash table\n");
 scanf("%d",&key);

 hkey=key%TABLE_SIZE;
 hash2 = 7-(key %7);

 for(i=0;i<TABLE_SIZE;i++)
 {
   index=(hkey+i*hash2)%TABLE_SIZE;
   if(h[index] == NULL)
   {
        h[index]=key;
        break;
   }
 }

 if (i == TABLE_SIZE)
     printf("\nelement cannot be inserted\n");
}

void search()
{

 int key,index,i,hkey,hash2;
 printf("\nenter search element\n");
 scanf("%d",&key);
 hkey=key%TABLE_SIZE;
 hash2 = 7-(key %7);

 for (i=0;i<TABLE_SIZE; i++)
 {
   index=(hkey+i*hash2)%TABLE_SIZE;
   if(h[index]==key)
   {
     printf("value is found at index %d",index);
     break;
   }
  }
```

```
      if (i == TABLE_SIZE)
         printf("\n value is not found\n");
}


void display()
{

     int i;

    printf("\nelements in the hash table are \n");
    for(i=0;i< TABLE_SIZE; i++)
        printf("\n Hash table [ %d ] =  %d",i,h[i]);

}

void main()
{
    int opt,i;
    clrscr();
    printf("DOUBLE HASHING\n");

   while(1)
    {
        printf("\nPress 1.Insert 2.Display 3.Search 4.Exit \n");
        scanf("%d",&opt);
        switch(opt)
        {
           case 1: insert();
                   break;
           case 2: display();
                   break;
           case 3: search();
                   break;
           case 4: exit(0);
        }
    }
}
```

Key density

The identifier density or key density of a hash table is the ratio n/T, where n is the number of identifiers in the table and T is the total number of possible keys.

Suppose our keys are at most six characters long, where a character may be a decimal digit or an upper case letter, and that the first character is a letter. Then the number of possible keys is $T = \sum_{0 \le i \le 5} 26x36^i > 1.6 * 10^9$. So the key density n/T is usually very small.

Loading density

The loading density or loading factor of a hash table is

$$\alpha = n / (sb)$$

Where n is the number of identifiers in the table
b is number of buckets
s is number of slots per bucket

Example :

Consider the hash table ht with buckets **b = 26** and slots **s = 2**.

We have **n = 10** distinct identifiers, each representing a C library function. This table has a loading factor, α, of 10/52 = 0.19.

$$\propto = \frac{n}{bs}$$

The hash function must map each of the possible identifiers onto one of the number, 0-25.

We can construct a fairly simple hash function by associating the letter, a-z, with the number, 0-25, respectively, and then defining the hash function, f (x), as the first character of x.

Using this scheme, the library functions acos, define, float, exp, char, atan, ceil, floor, clock, and ctime hash into buckets 0, 3, 5, 4, 2, 0, 2, 5, 2, and 2, respectively.

index	Slot 0	Slot 1
0	acos	atan
1		
2	char	ceil
3	define	
4	exp	
5	float	floor
6		
7		
8		
9		

25		

clock **ctime**

Table) Hash table with 26 buckets and two slots per bucket

The identifier clock hashes into the bucket ht[2]. Since this bucket is full, we have an overflow.

Dynamic Hashing

To ensure good performance, it is necessary to increase the size of a hash table whenever its loading density exceeds a prescribed threshold. So, for example if we currently have b buckets in our hash table and using the division hash function with divisor D=b. When an insert causes the loading density to exceed the pre-specified threshold, we use array doubling to increase the number of buckets to 2b.

2-010	10
4-100	00
5-101	01
3-011	11

00	4
01	5
10	2
11	3

Now if we want to insert any mode values in the hash table they will overflow. To avoid this we can use dynamic hashing. We can add some more memory and readjust the values already stored in the hash table and add the new values in the hash table.

Using 2 digited index there is a possibility of having 4 buckets. If we use 3 digited index there is a possibility of using 8 buckets. The storage of hash table will be doubled to allow you to store more values.

2-010
4-100
5-101
3-011
7-111
6-110

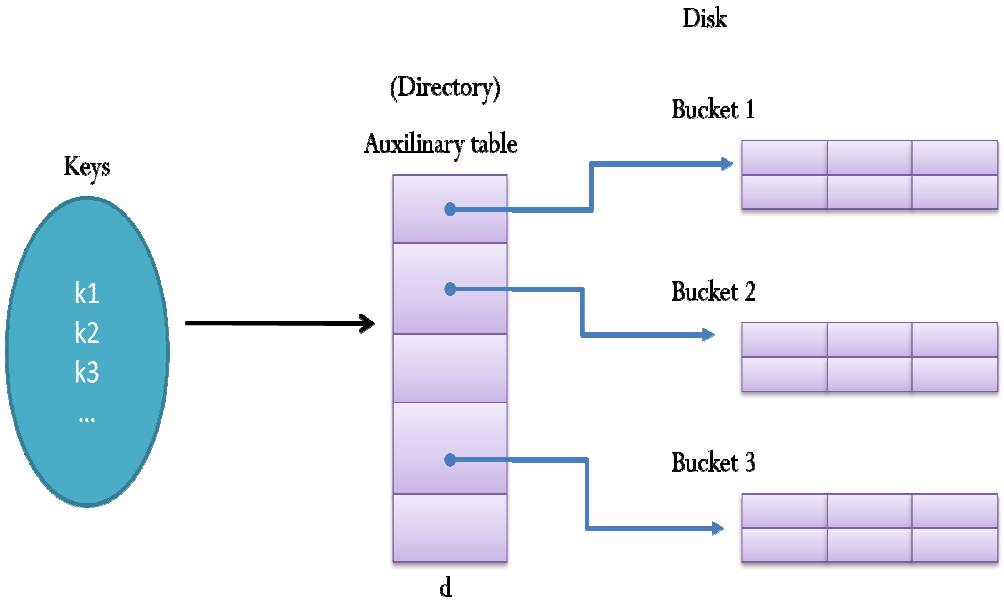
000	
001	
010	2
011	3
100	4
101	5
110	6
111	7

We consider two forms of Dynamic hashing- one uses a directory and the other does not.

1)Dynamic Hashing Using Directories

2)Directory Less Dynamic Hashing

Dynamic Hashing Using Directories



Dynamic Hashing

- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increases or decreases. This method is also known as Extendable hashing method.
- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

How to search a key

- First, calculate the hash address of the key.
- Check how many bits are used in the directory, and these bits are called as i.
- Take the least significant i bits of the hash address. This gives an index of the directory.
- Now using the index, go to the directory and find bucket address where the record might be.

How to insert a new record

- Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, then place the record in it.
- If the bucket is full, then we will split the bucket and redistribute the records.

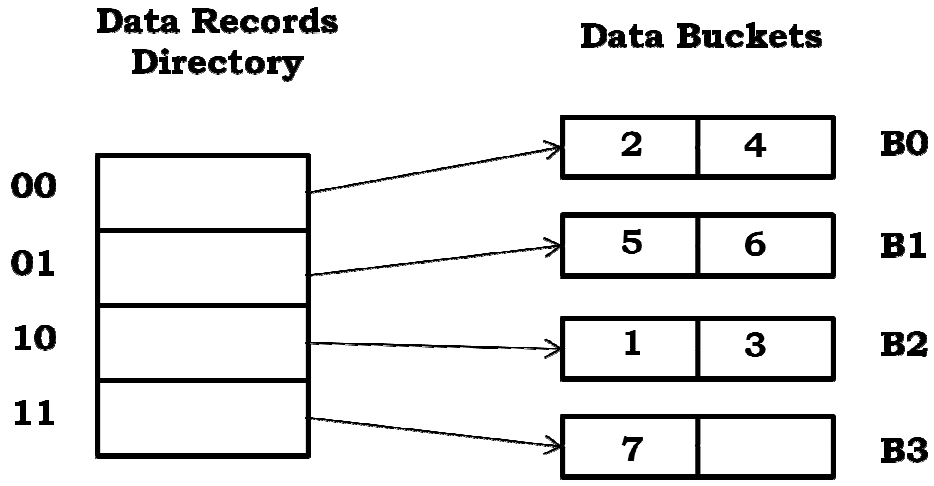
Example 1: Consider the following grouping of keys and insert them into buckets, depending on the prefix of their hash address:

Key	Hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111

Using two bits there is a possibility of producing 4 different codes. Assume that the directory has 4 codes and 4 buckets. Each bucket has 2 slots. Consider 00 pointing to Bucket B0, 01 pointing to Bucket B1, 10 pointing to bucket B2 and 11 pointing to bucket B3. Each bucket can store 2 values.

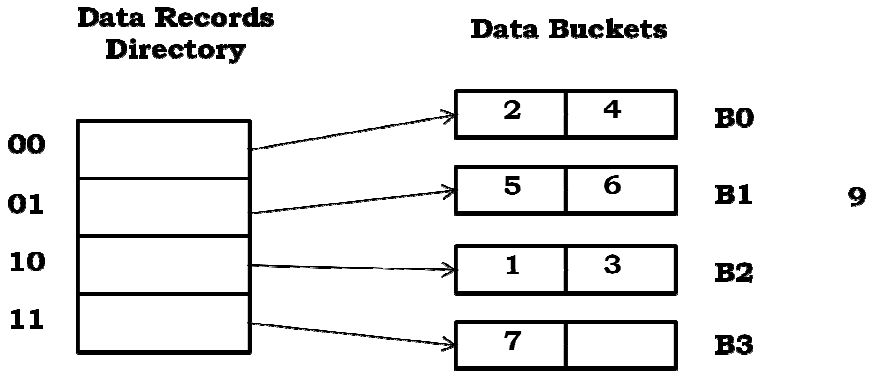
The last two bits of 2 and 4 are 00. So it will go into bucket B0. The last two bits of 5 and 6 are 01, so it will go into bucket B1. The last two bits of 1 and 3 are 10, so it will go into bucket B2. The last two bits of 7 are 11, so it will go into B3.

Key	Hash Address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111



Insert key 9 with hash address 10001 into the above structure:

- Since **key 9** has hash address **10001**, it must go into the bucket B1. But bucket B1 is full, so it will get split.



- The splitting will separate 5, 9 from 6 since last three bits of key 5 and key 9 are 001, so it will go into bucket B1, and the last three bits of key 6 are 101, so it will go into bucket B5.

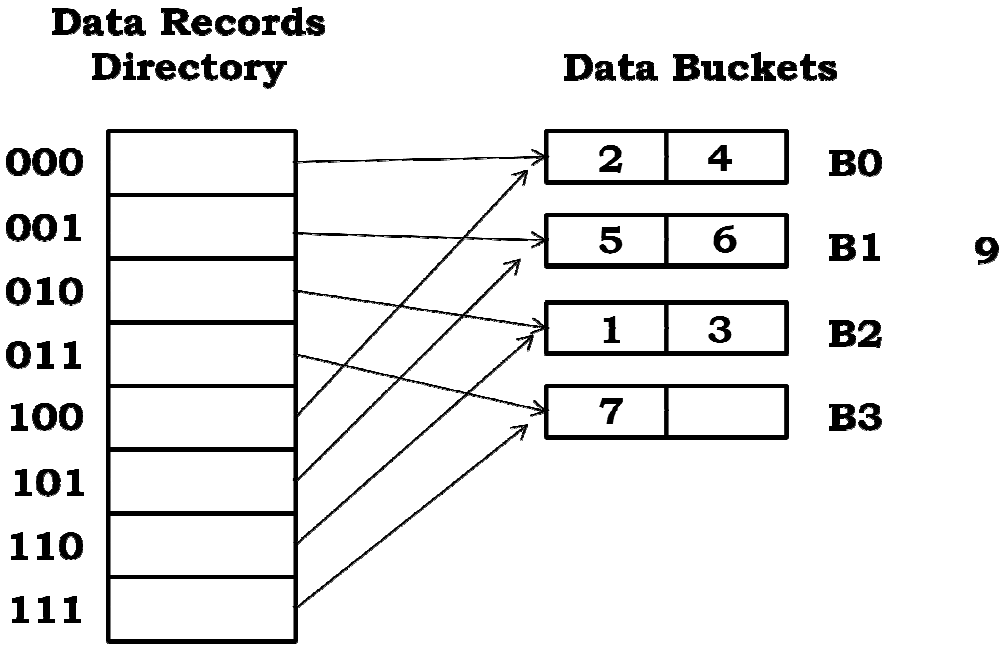
- Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00.

- Keys 1 and 3 are still in B2. The record in B2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.

- Key 7 are still in B3. The record in B3 pointed by the 111 and 011 entry because last two bits of both the entry are 11.

5 - 01001
6 - 10101
9 - 10001

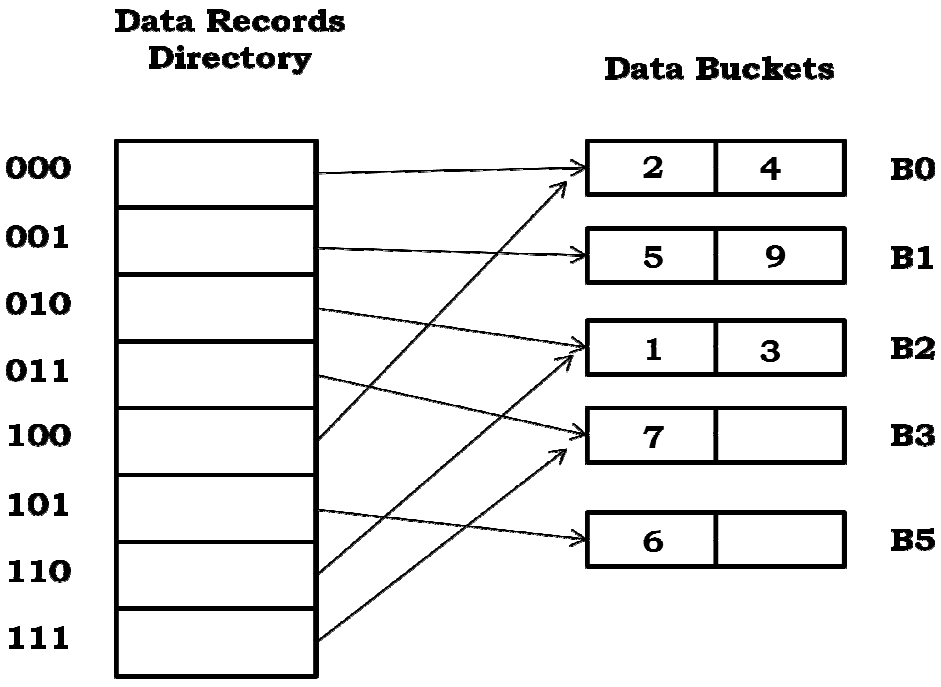
5 - 01001
6 - 10101
9 - 10001



5 - 01001
6 - 10101
9 - 10001

5 - 01001
6 - 10101
9 - 10001

B1
B5
B1



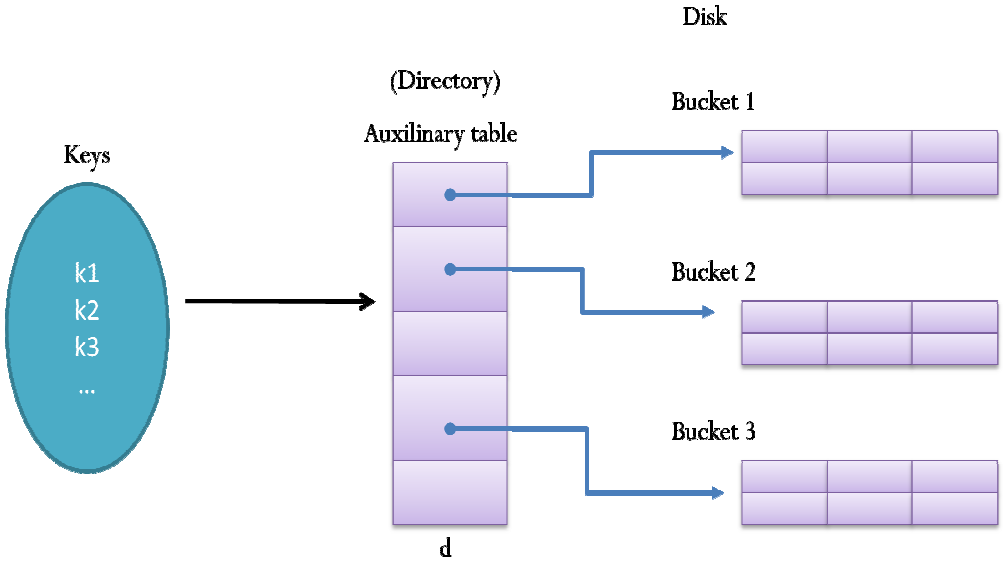
Advantages of dynamic hashing

- In this method, the performance does not decrease as the data grows in the system. It simply increases the size of memory to accommodate the data.
- In this method, memory is well utilized as it grows and shrinks with the data. There will not be any unused memory lying.
- This method is good for the dynamic database where data grows and shrinks frequently.

Disadvantages of dynamic hashing

- In this method, if the data size increases then the bucket size is also increased. These addresses of data will be maintained in the bucket address table. This is because the data address will keep changing as buckets grow and shrink. If there is a huge increase in data, maintaining the bucket address table becomes tedious.
- In this case, the bucket overflow situation will also occur. But it might take little time to reach this situation than static hashing.

Dynamic Hashing Using Directories Uses an auxiliary table to record the pointer of each bucket



Example 2) Define the hash function h(k) transforms k into 6-bit binary integer. Insert keys in the hash table using dynamic hashing using directory.

k	h(k)
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C2	110 010
C3	110 011
C5	110 101
C1	110 001
C4	110 100

The size of directory d is 2^r, where r is the number of bits used to identify all h(k).

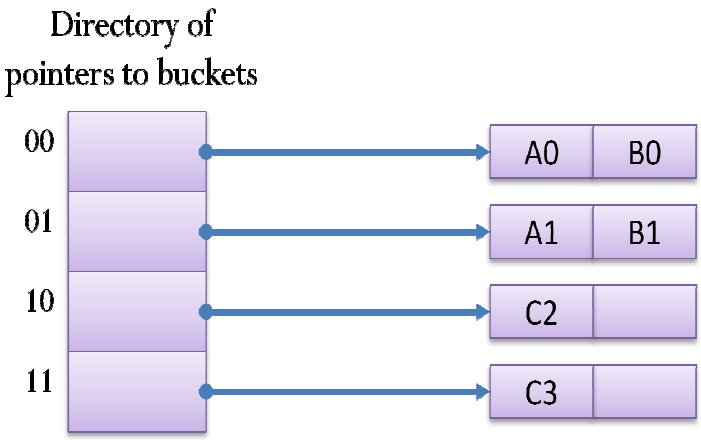
Initially, Let r = 2. Thus, the size of directory d = 2^2 = 4.

Suppose h(k, p) is defined as the p least significant bits in h(k), where p is also called dictionary depth.
E. g.

h(C5) = 110 101
h(C5, 2) = 01
h(C5, 3) = 101

Consider the following keys have been already stored. The least r is 2 to differentiate all the input keys.

k	h(k)
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C2	110 010
C3	110 011



Depth 2

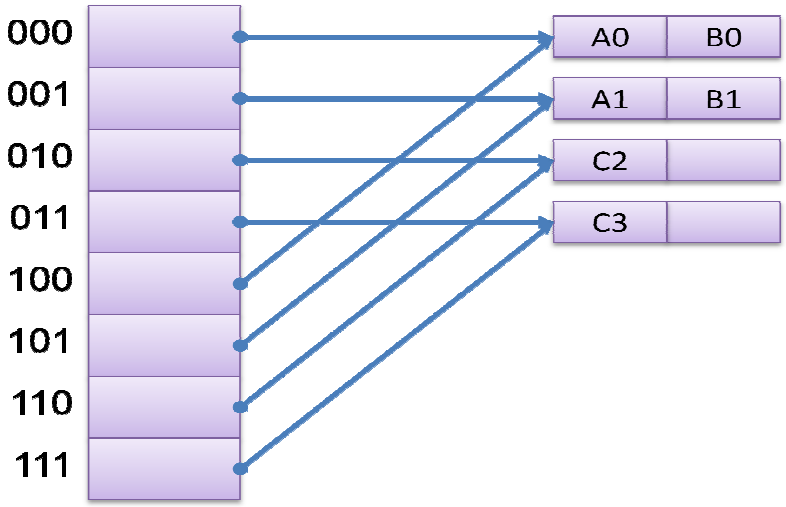
When C5 (110101) is to enter in to the buckets of the hash table, we normally consider the least 2 digits of the hash address which is **01**

Key	Hash address
A1 -	100001
B1 -	101001
C5 -	110101

In the directory we can see this is pointing to a bucket where both slots are already filled with A1 and B1. The Key C5 is over flow from This bucket. Now let us consider the least 3 bits of the hash address.

Key	Hash Address
A1	100 001
B1	101 001
C5	110 101

If we consider 3 bits as directory index we can have 8 pointers in the directory.



Depth 3

When C5 (110101) is to enter

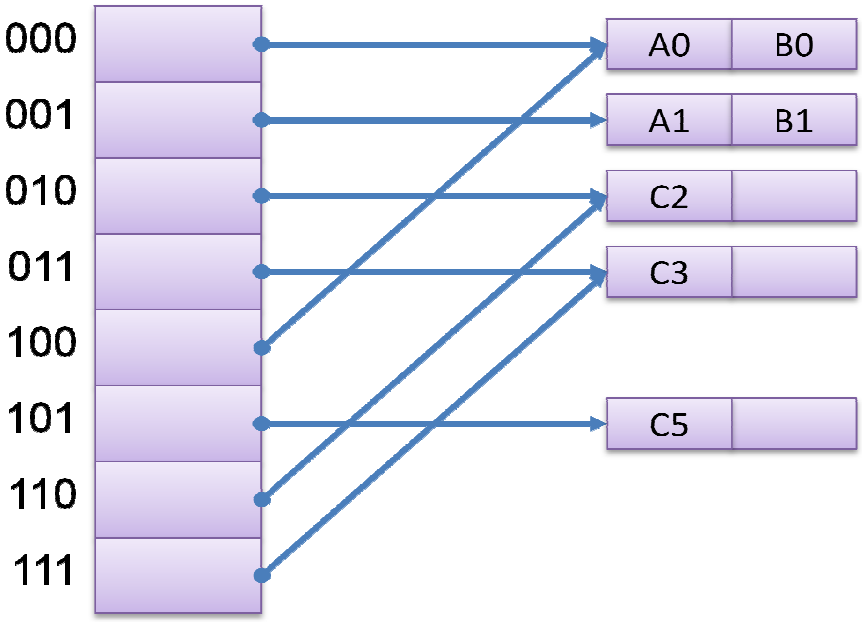
Since r=2 and h(C5, 2) = 01, follow the pointer of d[01].

A1 and B1 have been at d[01]. Bucket overflows.

Find the least u such that h(C5, u) is not the same with some keys in h(C5, 2) (01) bucket.
In this case, u = 3.

Since u > r, expand the size of d to 2^u and duplicate the pointers to the new half (why?).

Rehash identifiers 01 (A1 and B1) and C5 using new hash function h(k, u).



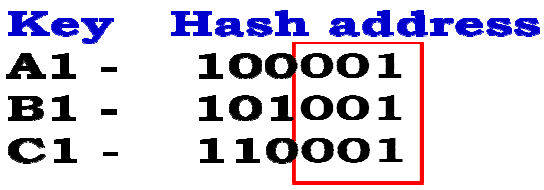
Depth 3

Let r = u = 3.

When **C1 (110001)** is to enter

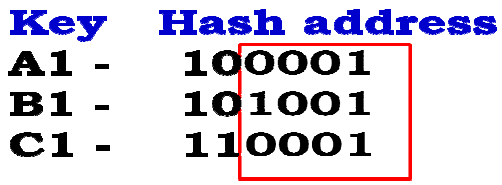
Since r=3 and h(C1, 3) = 001, follow the pointer of d[001].

A1 and B1 have been at d[001]. Bucket overflows.



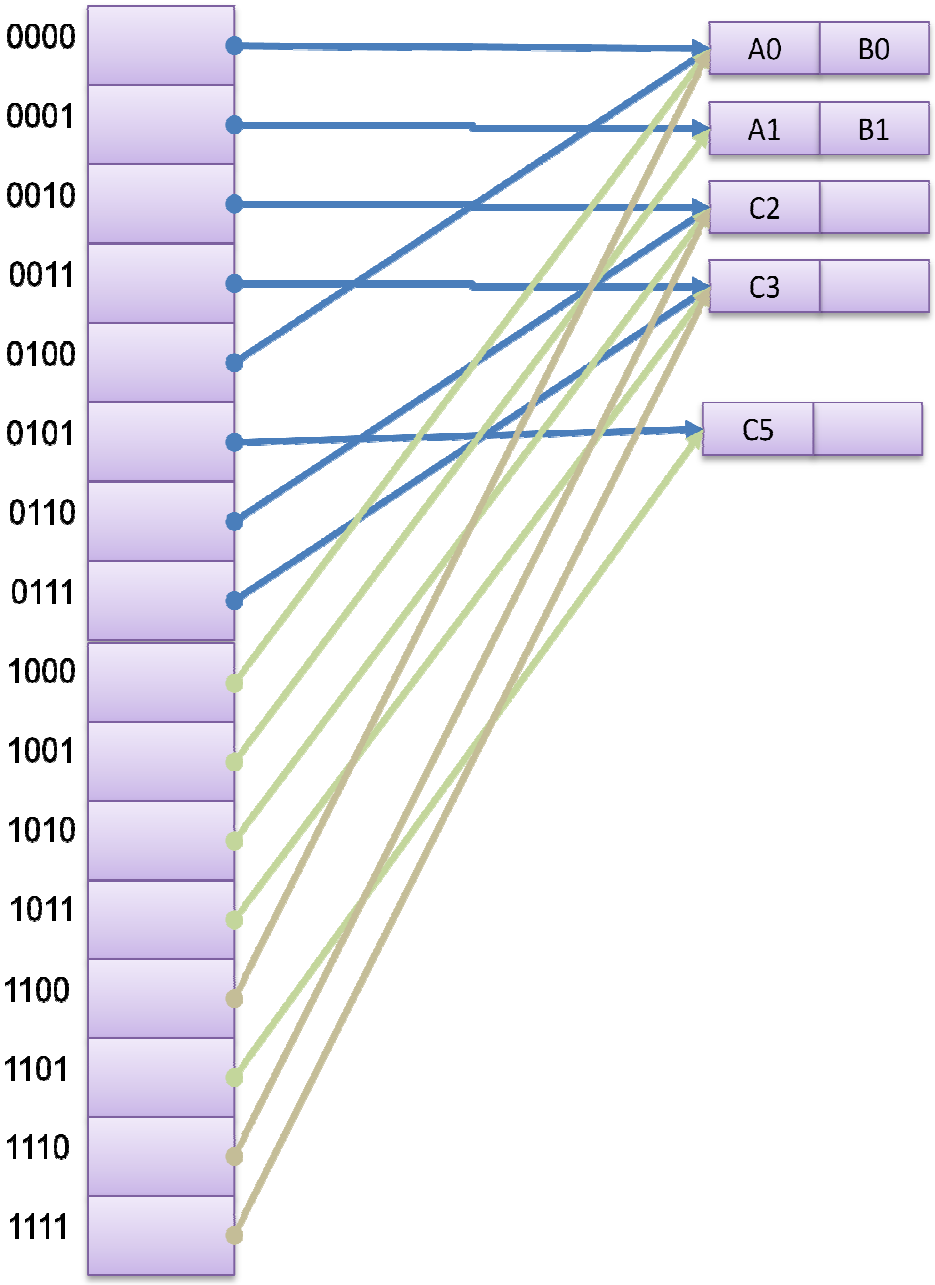
Find the least u such that h(C1, u) is not the same with some keys in h(C1, 3) (001) bucket.

In this case, u = 4.

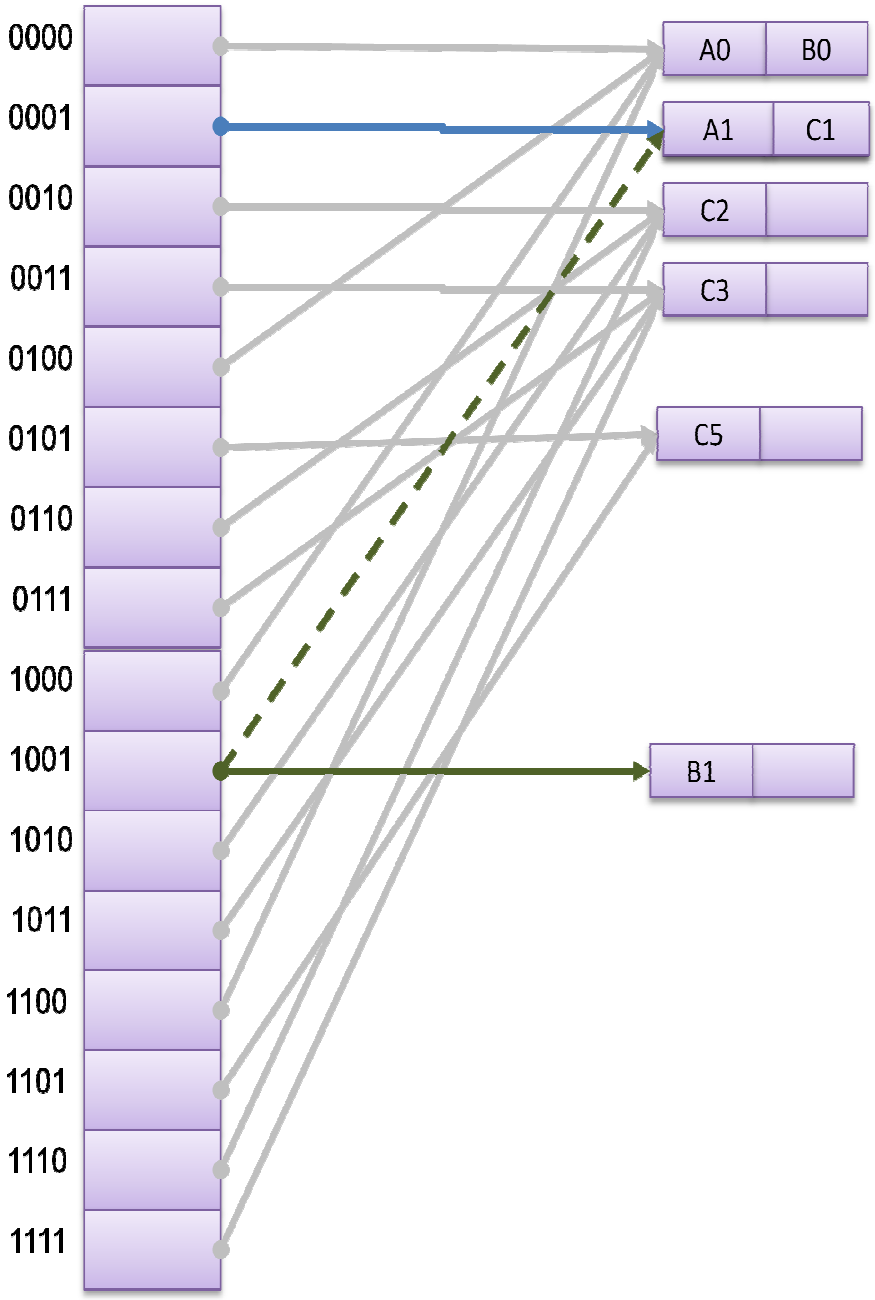


A1, C1 will be placed in one bucket and B1 will be placed in new bucket.

Since u > r, expand the size of d to 2^u and duplicate the pointers to the new half.


0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
A0
B0
A1
B1
C2
C3
C5

Depth 4


0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
A0
B0
A1
C1
C2
C3
C5
B1

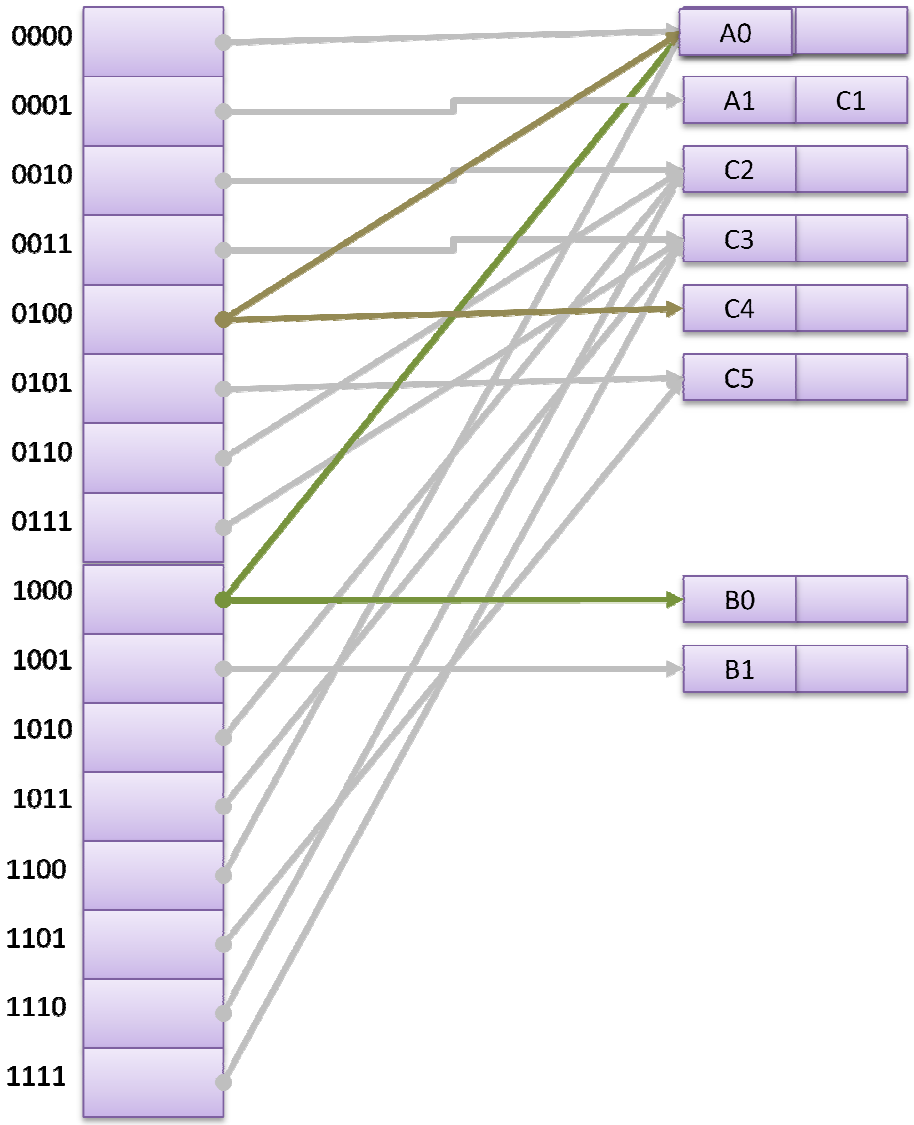
depth 4

When C4 (110100) is to enter

Since r=4 and h(C4, 4) = 0100, follow the pointer of d[0100].
A0 (100000) and B0 (101000) have been at d[0100]. Bucket overflows.

A0----100000
B0----101000
C1----110001
C4----110100

Find the least u such that h(C1, u) is not the same with some keys in h(C1, 4) (0100) bucket.
In this case, u = 3.
Since u = 3 < r = 4, d is not required to expand its size.



Advantages

- Only doubling directory rather than the whole hash table used in static hashing.
- Only rehash the entries in the buckets that overflows.

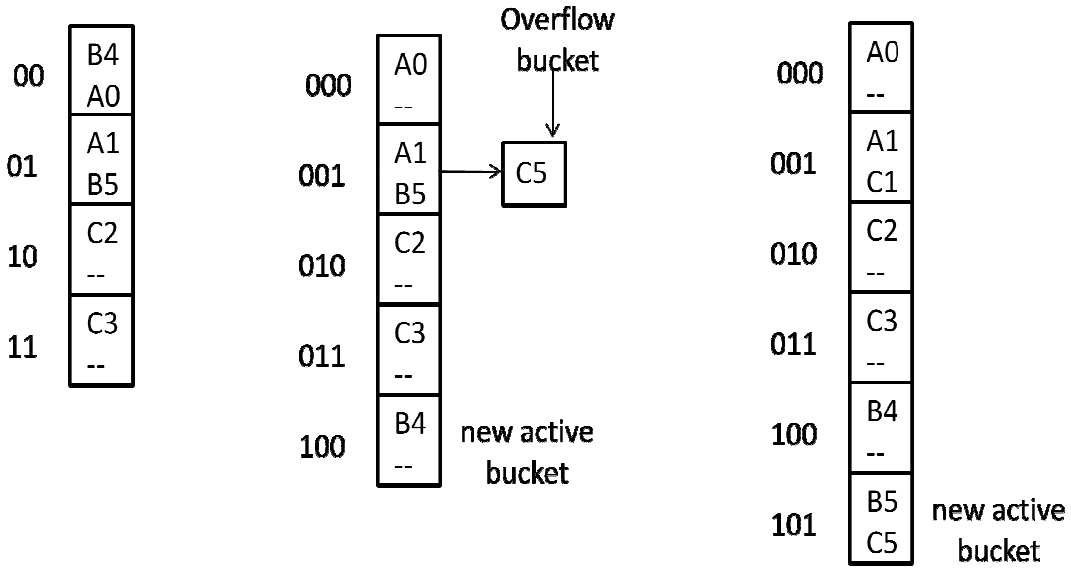
Directory Less Dynamic Hashing

Directory less Dynamic hashing is also known as linear dynamic hashing

k	h(k)
A0	100 000
A1	100 001
B0	101 000
B1	101 001
B4	101 100
B5	101 101
C1	110 001
C2	110 010
C3	110 011
C5	110 101

Figure a) shows a directory less hash table ht with r=2 the number of bits of h(k) used to index into the hash table and q=0. The number of active buckets are 4 indexed (00 01 10 11). Each active bucket has 2 slots.

Insert B4, A0, A1, B5, C2 and C3. Each dictionary pair is either in an active or an overflow bucket.



a) r=2, q=0 b)insert C5, r=2, q=1 c) insert C1. r=2,q=2

d is directory buckets=2^r

additional links q may be added where $0<=q<=2^r$.

Figure b) r=2, q=1, h(k,3) has been used for chains 000 and 100.
Insert C5
q =1 means one bucket added
h(k,3) means using lower 3 digits as index.
after adding a bucket readjust keys.
C5 is in overflow bucket

Figure c) r=2 , q=2, h(k,3) has been used for chains 000 , 100 and 101.
Insert C1, q=2 means 2 buckets added

An example of directory less hashing after two insertions.

Initially, there are four pages/buckets, each addressed by two bits (Figure (a). Two of the pages/buckets are full, and two have one identifier each.

When C5 is inserted, it hashes to the page/bucket whose address is 01 (Figure (b). Since that page/bucket is full, an overflow node is allocated to hold C5. At the same time, we add a new page at the end of the storage, rehash the identifiers in the first page, and split them between the first and new page. B4 moves to last page as the last 3 bits of B4 are 100. The last 3 bits of key C5 are 101. And we don't have a bucket with address 101 so C5 still in overflow node.

In the next step, we insert the identifier C1. Last 3 bits of C1 are 001. Since it hashes to the same page as C5, we use another overflow node to store it.

A1 - 100 001
B5 - 101 101
C5 - 110 101
C1 - 110 001

We add another new page to the end of the file and rehash the identifiers in the second page. The address of new bucket added to the has table is 101.

Last 3 bits of A1, C1 are 001. Last 3 Bits of B5 and C5 are 101. So B5 and C5 will be storing in the new bucket and C1 will be storing in 001 bucket so that there will be no overflow.

QUESTIONS FROM PREVIOUS UNIVERSITY EXAMINATIONS

1) List some applications of Hashing

Hashing provides constant time search, insert and delete operations on average. This is why hashing is one of the most used data structure, example problems are, distinct elements, counting frequencies of items, finding duplicates, etc. There are many other applications of hashing, including modern day cryptography hash functions. Some of these applications are listed below:

- Message Digest
- Password Verification
- Data Structures(Programming Languages)
- Compiler Operation
- Rabin-Karp Algorithm
- Linking File name and path together

2) With suitable examples, discuss about the hash functions: mid-square, folding and digit analysis.

Page No's 6,7 & 8

3) Write and explain procedure to delete a dictionary pair from a directory less dynamic hash table.

Page 32

4) Define Hash table and Hash function

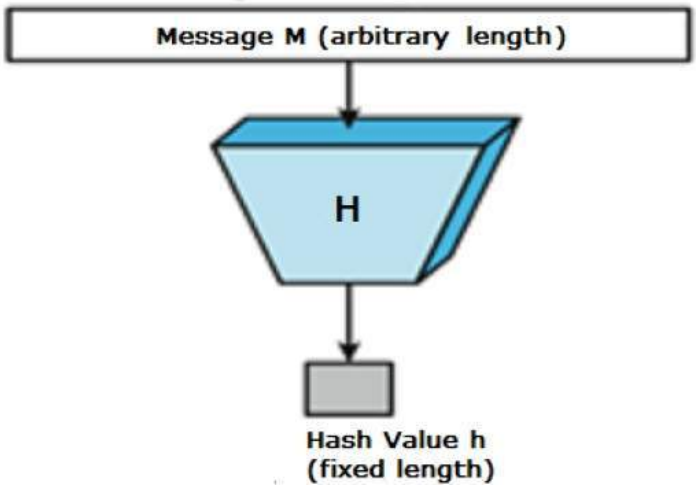
Page 3

5) Write short notes on secure Hash functions

Hash functions are extremely useful and appear in almost all information security applications.

A hash function is a mathematical function that converts a numerical input value into another compressed numerical value. The input to the hash function is of arbitrary length but output is always of fixed length.

Values returned by a hash function are called **message digest** or simply **hash values**. The following picture illustrated hash function –



6) Discuss about dynamic hashing with directories. Give an example

Page no's 20,21,22,23,24

7) List pros and cons of chaining and open addressing

A chained hash table indexes into an array of pointers to the heads of linked lists. Each linked list cell has the key for which it was allocated and the value which was inserted for that key. When you want to look up a particular element from its key, the key's hash is used to work out which linked list to follow, and then that particular list is traversed to find the element that you're after. If more than one key in the hash table has the same hash, then you'll have linked lists with more than one element.

An open-addressing hash table indexes into an array of pointers to pairs of (key, value). You use the key's hash value to work out which slot in the array to look at first. If more than one key in the hash table has the same hash, then you use some scheme to decide on another slot to look in instead. For example, linear probing is where you look at the next slot after the one chosen, and then the next slot after that, and so on until you either find a slot that matches the key you're looking for, or you hit an empty slot (in which case the key must not be there).

Open-addressing is usually faster than chained hashing when the load factor is low

8) Define a dictionary. Give few examples for dictionaries.

A *dictionary* is a general-purpose data structure for storing a group of objects. A dictionary has a set of *keys* and each key has a single associated *value*. When presented with a key, the dictionary will return the associated value. Hash table is nothing but a dictionary.

Dictionaries typically support several operations:

- retrieve a value
- insert or update a value
- remove a key-value pair
- test for existence of a key

9) Show the hash function h(k)=k%17 does not satisfy the one way property, weak collision resistance and strong collision resistance.

Using hash function we are able to find index of k. We can store k in the hash table at position index.

We are unable to find k using the hash without hash table.

H(153)=153%17=0

Using 0 we can't get 153 without hash table. So the hash function is not satisfying one way property.

Collision Resistance : no two elements with the hash function should generate the same hash value.

10) Write and explain procedure to insert a dictionary pair into a dynamic hash table that uses a directory.

Page no's 20,21,22,23,24

11) Define key density and loading density. Explain with example.

Key density

The identifier density or key density of a hash table is the ratio n/T, where n is the number of identifiers in the table and T is the total number of possible keys.

Suppose our keys are at most six characters long, where a character may be a decimal digit or an upper case letter, and that the first character is a letter. Then the number of possible keys is $T = \sum_{0 \le i \le 5} 26x36^i > 1.6 * 10^9$. So the key density n/T is usually very small.

Loading density

The loading density or loading factor of a hash table is

$$\alpha = n / (sb)$$

Where n is the number of identifiers in the table
b is number of buckets
s is number of slots per bucket

Example :

Consider the hash table ht with buckets **b = 26** and slots **s = 2**.

We have **n = 10** distinct identifiers, each representing a C library function. This table has a loading factor, α, of 10/52 = 0.19.

The hash function must map each of the possible identifiers onto one of the number, 0-25.

We can construct a fairly simple hash function by associating the letter, a-z, with the number, 0-25, respectively, and then defining the hash function, f (x), as the first character of x.

Using this scheme, the library functions acos, define, float, exp, char, atan, ceil, floor, clock, and ctime hash into buckets 0, 3, 5, 4, 2, 0, 2, 5, 2, and 2, respectively.

index	Slot 0	Slot 1
0	acos	atan
1		
2	char	ceil
3	define	
4	exp	
5	float	floor
6		
7		
8		
9		

25		

clock **ctime**

Hash table with 26 buckets and two slots per bucket

The identifier clock hashes into the bucket ht[2]. Since this bucket is full, we have an overflow.

12) With suitable example explain about linear probing, quadratic probing and rehashing.

Linear probing --- page 11
Quadratic probing --- page 13

Rehashing:

As the name suggests, **rehashing means hashing again**. Basically, when the load factor increases to more than its pre-defined value (default value of load factor is 0.75), the complexity increases. So to overcome this, the size of the array is increased (doubled) and all the

values are hashed again and stored in the new double sized array to maintain a low load factor and low complexity.

Why Rehashing?

Rehashing is done because whenever key value pairs are inserted into the map, the load factor increases, which implies that the time complexity also increases as explained above. This might not give the required time complexity of O(1).

Hence, rehash must be done, increasing the size of the bucketArray so as to reduce the load factor and the time complexity.

13) Write and explain procedure to insert a dictionary pair into a dynamic hash table that uses a directory.

Page no's 20,21,22,23,24

14) What are collision and overflow with respect to hashing

What is Collision?

Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

15) Write and explain procedure to delete a dictionary pair from a dynamic hash table that uses a directory.

Page no's 20,21,22,23,24

16) Let α=n/b be the loading density of a uniform hashing function h. Then derive expressions for the expected number of key comparisons U_n and the average number of key comparisons S_n for linear open addressing and for chaining.

Let α=n/b be the loading density of a hash table using a uniform hashing function h. Then for linear open addressing

$$U_n \approx \frac{1}{2}[1+\frac{1}{(1-\alpha)^2}] \qquad S_n \approx \frac{1}{2}[1+\frac{1}{1-\alpha}]$$

for rehashing, random probing, and quadratic probing

$$U_n \approx 1/(1-\alpha) \qquad S_n \approx -[\frac{1}{\alpha}]\log_e(1-\alpha)$$

for chaining

$$U_n \approx \alpha \qquad S_n \approx 1+\alpha/2$$

17) Define open addressing

Open addressing, or **closed hashing**, is a method of collision resolution in hash tables. With this method a hash collision is resolved by **probing**, or searching through alternate locations in the array (the *probe sequence*) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.[1] Well-known probe sequences include:

Linear probing
in which the interval between probes is fixed — often set to 1.

Quadratic probing
in which the interval between probes increases quadratically (hence, the indices are described by a quadratic function).

Double hashing
in which the interval between probes is fixed for each record but is computed by another hash function.

The main tradeoffs between these methods are that linear probing has the best cache performance but is most sensitive to clustering, while double hashing has poor cache performance but exhibits virtually no clustering; quadratic probing falls in-between in both areas. Double hashing can also require more computation than other forms of probing.

18) Explain folding Hashing techniques with suitable examples.

Shift folding

Suppose that k = 12320324111220 and we partition it into parts that are three decimal digits long. The partitions are P1=123 P2= 203 P3=241 P4=112 P5=20 using shift folding we obtain

$$h(k) = \sum_{i=1}^{5} P_i = 123 + 203 + 241 + 112 + 20 = 699$$

Folding at the boundaries.

Alternate partitions reversed
h(k)=123+302+241+211+20=897

19) Discuss the techniques for insert a directory pair into a dynamic hash table that uses a directory.

Page no's 20,21,22,23,24

20) Write an algorithm to delete a directory pair from a directory less dynamic hash table.

Page 32

21) Explain the digit analysis Hashing technique with suitable examples.

Page 8

22) Discuss the techniques for deleting a directory pair into a dynamic hash table that uses a directory.

Page no's 20,21,22,23,24

23) Write an algorithm to insert a directory pair from a directory less dynamic hash table.

Page 32

24) What are hash functions? List some techniques that are used to implement Hash functions.

a) Division Method
b) Mid square Method
c) Digit folding Method
d) Digit Analysis Method/Binary/Radix method

25) What is rehashing

As the name suggests, **rehashing means hashing again**. Basically, when the load factor increases to more than its pre-defined value (default value of load factor is 0.75), the complexity increases. So to overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double sized array to maintain a low load factor and low complexity.

Why rehashing?

Rehashing is done because whenever key value pairs are inserted into the map, the load factor increases, which implies that the time complexity also increases as explained above. This might not give the required time complexity of O(1).

Hence, rehash must be done, increasing the size of the bucketArray so as to reduce the load factor and the time complexity.

26) What do you mean by hashing? Why do we need it?

Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value.

27) The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function h(k)=k mod 10 and linear probing. What is the resultant hash table.

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

28) Explain about hash table restructuring with examples.

When a hash table has a large number of entries (i.e., let us say $n >= 2m$ in open hash table or $n >= 0.9m$ in closed hash table), the average time for operations can become quite substantial. In such cases, one idea is to simply create a new hash table with more number of buckets (say twice or any appropriate large number).

In such a case, the currently existing elements will have to be inserted into the new table. This may call for

a. rehashing of all these key values
b. transferring all the records

This effort will be less than it took to insert them into the original table.

Subsequent dictionary operations will be more efficient and can more than make up for the overhead in creating the larger table.

29) List the methods of hash function.

a) Division Method
b) Mid square Method
c) Digit folding Method
d) Digit Analysis Method/Binary/Radix method

30) Explain about the analysis of closed hashing for successful search and deletion.

Load factor α

α = n/m = no of elements stored in the table/total no of elements in the table

Assume uniform hashing. In this scheme, the probe sequence

$< h(k, 0),..., h(k, m - 1) >$

for each key k is equally likely to be any permutation of $< 0, 1,..., m - 1 >$

31) Explain different collision resolution strategies for hashing. State the advantages and disadvantages of each technique.

a) Chaining
b) Linear Probing
c) Quadratic Probing and
d) Double Hashing.

32) What are primary and secondary clustering problems? Suggest some open addressing methods to avoid them.

Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

Delete(k): ***Delete operation is interesting***. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".

Insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

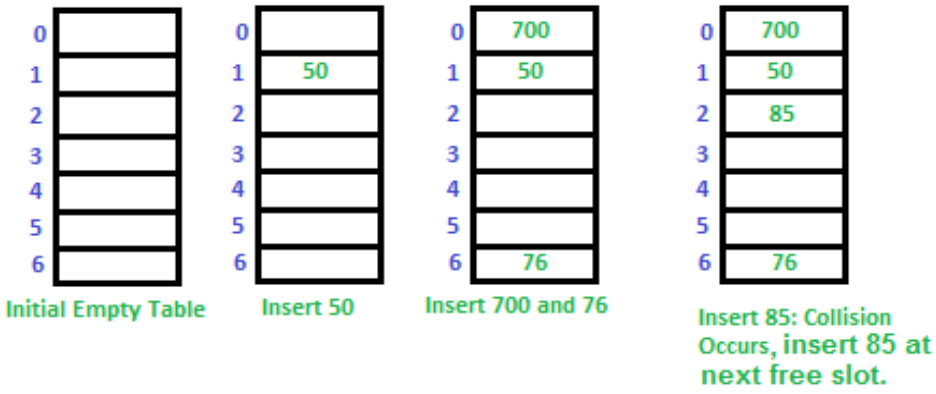
Open Addressing is done in following ways:

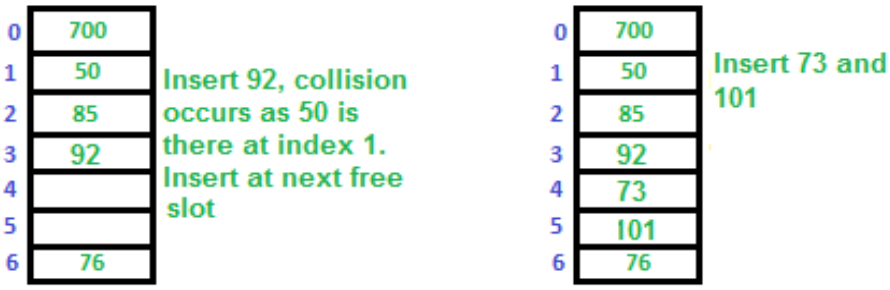
a) Linear Probing: In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

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.





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.

b) ***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

c) Double Hashing We use another hash function hash2(x) and look for i*hash2(x) slot in i’th rotation.

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

Comparison of above three:

Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.

Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

S.No.	Separate Chaining	Open Addressing
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care for to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

Performance of Open Addressing:

Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing)

m = Number of slots in the hash table

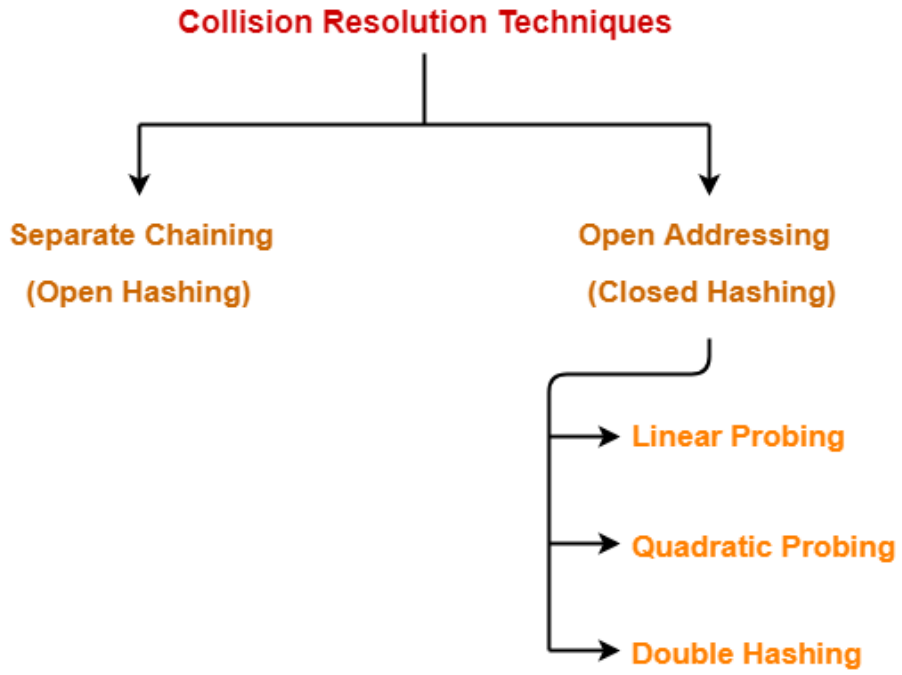
n = Number of keys to be inserted in the hash table

Load factor $\alpha = n/m \quad (< 1)$

Expected time to search/insert/delete $< 1/(1 - \alpha)$

So Search, Insert and Delete take $(1/(1 - \alpha))$ time

33) What are different collision resolution techniques



In open addressing, all the keys are stored inside the hash table. No key is stored outside the hash table. The Techniques used for open addressing are-

a) **Linear Probing**
b) **Quadratic Probing**
c) **Double Hashing**

Closed addressing is the traditional approach, which solves collisions by allowing more than one element in each bucket. One way to do closed addressing is "chaining", where each bucket stores a linked list of elements in that bucket.