Dictionaries

Definition:

A dictionary is a general purpose data structure which is an ordered or unordered list of key-value pairs, where keys are used to locate values in the list. Each value in a dictionary is associated with the unique key.

Dictionaries can be ordered or unordered, but in most of the real-time application, we choose ordered i.e., sorted dictionaries

Examples:

- Consider a data structure that stores bank accounts, it can be viewed as a dictionary, where account no.'s serve as keys for identification of accounts.
- Consider a data structure that stores student data, it can be viewed as a dictionary, where student Id serve as key for identification of student's information.
- Consider a data structure for phone directory; it can be viewed as a dictionary, where person names serve as keys to identify phone numbers

Applications of Dictionaries:

There are various applications of dictionaries. Some of them are:

- Symbol - table of a Compiler,
- Routing tables for network communication,
- Page tables,
- Associative arrays (Python),
- Spell Checkers,
- Document fingerprints, etc.,

Operations on Dictionaries:

Dictionaries typically support several operations such as:

- Insert (if the key does not exist in the dictionary the key-value pair is inserted, else its corresponding old value is overwritten with the new value).
- Delete (Key-value pair associated with given key is deleted).
- Retrieve (Key-value pair associated with given key is displayed / retrieved).
- Display (All key-value pairs are displayed)

- Search (tests the existence of a given key)
- Size or Length (count of key-value pairs)

Implementation of Dictionaries:

Dictionary is an abstract data type which can be implemented using

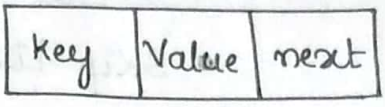
- Arrays
- Linked Lists
  - Sorted List
  - Skip List
- Hashing
- Trees
  - Binary Search Trees
  - Balanced Search Trees
    - AVL Trees
    - Red Black Trees
    - Splay Trees
  - Multi-way Search Trees
  - Tries

Linear List or Sorted List Implementation of Dictionary:

In linear list representation of dictionary, each node consists of three parts i.e., key, value and a pointer to the next node.

Each node can be defined using following structure:

```
struct dict
{
    datatype key;
    datatype value;
    struct dict *next; //*link
} * start = NULL; → it indicates that dictionary is empty.
```



Basically, we use int datatype.

The nodes should be in sorted (ascending) order with respect to the keys associated with the values.

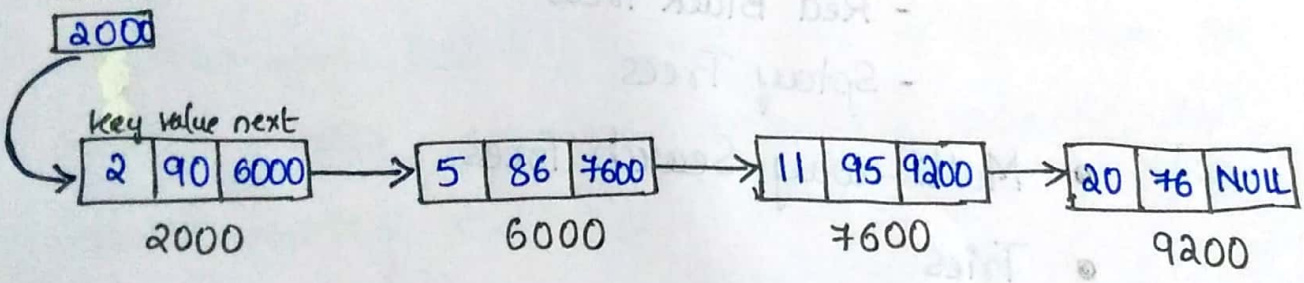


Fig: Logical Representation of Dictionary using sorted list

Operations on dictionaries using linear list :

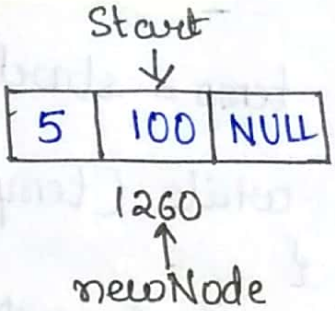
<1> Insert

Every new node should be inserted into a dictionary in such a way that the nodes should remain in sorted order with respect to keys.

If the given key (to be inserted) is already present in dictionary, then its associated value should be modified / update ; else the given key-value pair is inserted in to a dictionary.

1) Inserting First Node :

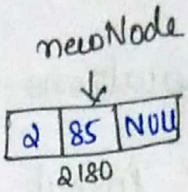
```
if (start == NULL)
    start = newNode;
```

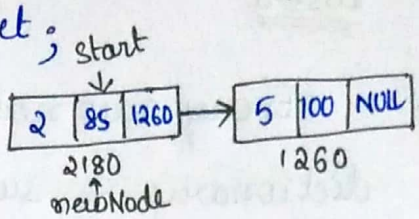


As key-value pairs (nodes) are to inserted in sorted order, we use following criteria to insert every new node

2) If the key of new node is less than the key of the start node :

```
if (newNode -> Key < start -> key)
{
    newNode -> next = start;
    start = newNode;
}
```

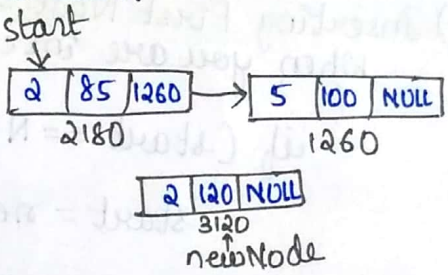


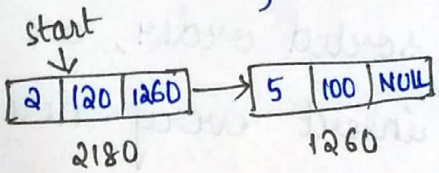


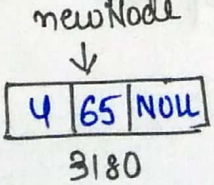
3) Inserting a new node in between the nodes of dictionary:

In this case, we have to check whether the key is already present in dictionary or not, if yes we have to modify its old value with new value else we have to insert node such that they are in sorted order.

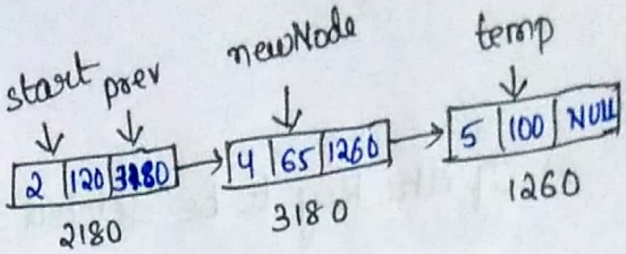
```
temp = start;
while (temp != NULL)
{
    if (newNode -> Key == temp -> key)
    {
        temp -> value = newNode -> value;
        free (newNode);
        break;
    }
    else if (newNode -> key < temp -> key)
    {
        prev -> next = newNode;
        newNode -> next = temp;
    }
    break;
}
```





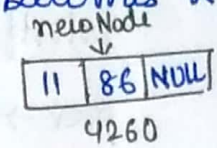


prev = temp;
temp = temp → next;
}

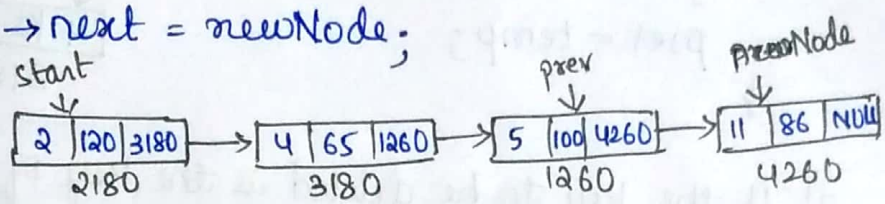


4) Inserting a new node whose key is greater than the key of last node:

After criteria 3 checking temp becomes NULL and prev points to last node.




```
if (temp == NULL && newNode → key > prev → key)
{
    prev → next = newNode;
}
```

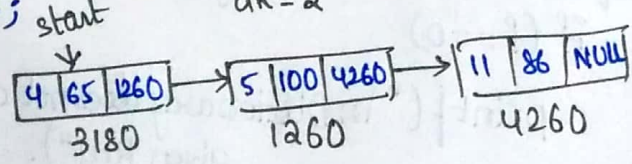


<2> Delete

If the given key is present in dictionary, it deletes key-value pair else displays that key not present in dictionary; For this, initialize f = 0

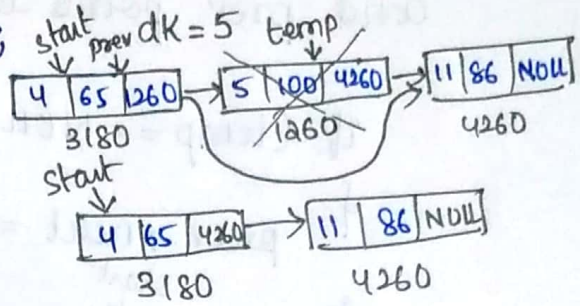
1) If the key to be deleted is the key of start node:

```
if (start → key == dK)  //dK → the key to delete
{
    f = 1;
    printf("\n %d-%d pair is deleted", start → key, start → value);
    ptr = start → next;
    free(start);
    start = ptr;
}
```



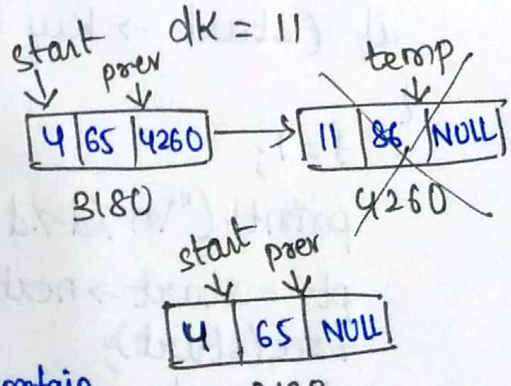
2) If the key to be deleted is a key of intermediate node:

```
for (temp=start; temp!=NULL; temp=temp->next)
{
  if (temp->key==dk)
  {
    f=1;
    printf("\n%d-%d pair is deleted", temp->key, temp->value);
    prev->next=temp->next;
    free(temp);
  }
  prev=temp;
}
```



3) If the key to be deleted is the key of lastnode:

```
for (temp=start; temp!=NULL; temp=temp->next)
{
  if (temp->key==dk && temp->next==NULL)
  {
    f=1;
    printf("\n%d-%d pair is deleted", temp->key, temp->value);
    prev->next=NULL;
    free(temp);
  }
  prev=temp;
}
```



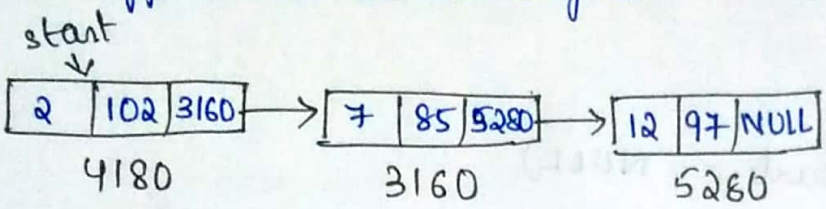
If still after checking

```
if (f==0)
  printf("\nDictionary does not contain given key");
```

<3> Display

It displays all key-value pairs present in the dictionary, if dictionary is not empty.

Suppose the dictionary is



then the output is

Key - value Pairs are..

2 - 102

7 - 85

12 - 97

```
if (start == NULL)
    printf ("\n Dictionary is Empty");
else
{
  printf ("\n Node values are : \n");
  for (temp = start; temp != NULL ; temp = temp→next)
     printf ("\n %d - %d", temp→key, temp→value);
}
```

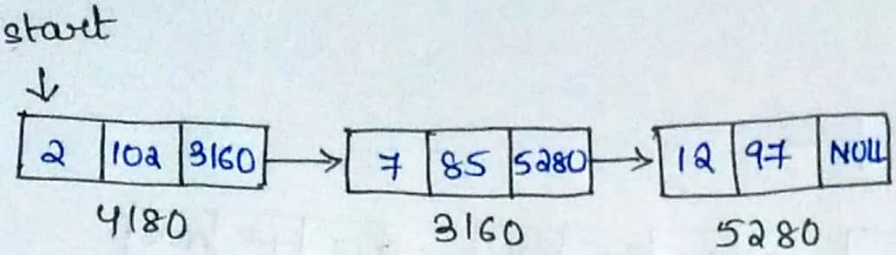
<4> Retrieve

If the given key is present in dictionary, it displays the key-value pair, else displays that key not present in dictionary; For this, initialize f=0

```
if (start == NULL)
    printf("\n Dictionary is Empty");
else
{
 for (temp = start; temp != NULL; temp = temp->next)
 {
   if (temp->key == rk)
   {  f=1;
      printf("\n Key-Value : %d-%d", temp->key, temp->value);
      break;
   }
 }
}
```

After checking all keys of dictionary, still

```
if (f==0)
   printf("\n Dictionary does not contain given key");
```



If key to be retrieved, rk = 7 then

output is

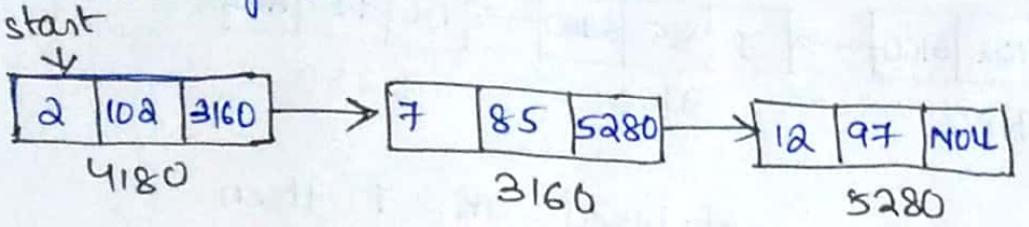
Key - Value : 7 - 85

<5> Length / Size

If the dictionary is not empty, then it displays the count of key-value pairs present in the dictionary.

```
int c = 0;
if (start == NULL)
    printf ("\n Dictionary is Empty");
else
{
  for (temp = start; temp != NULL; temp = temp -> next)
      c++;          // counts no. of Key - value pairs
}
printf ("\n No. of. Key-value pairs present in
        dictionary : %d", c);
```

If dictionary is



then the output is

No. of key-value pairs present in dictionary : 3

Hashing:

The two Search algorithms: linear Search and binary Search. Linear Search has a running time proportional to O(n), while binary Search takes time proportional to O(logn). where n is the number of elements in the array. Binary Search and Binary Search trees are efficient algorithms to Search for an element. But if we want to perform the Search operation in time proportional to O(1) then we have a Concept Called Hashing

Hashing is an effective way of storing and retrieving the elements to/from a data structures. There are two Concepts used regarding hashing and those are

Hash Tables.

Hash Functions.

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/element value. Hence every entry in the hash table is associated with some key.

Hash Function:

A Hash Function is a mathematical formula which applied to a key, produces an integer which can be used as an index for the key or element in the hash table.

The main aim of a hash function is that elements should be relatively, randomly and uniformly distributed.

Hashing Methods:

There are various types of hash functions that are used to place records in the hash table:

They are:

Division Method

Mid Savuare Method

Multiplicative Method

Digit folding Method.

Division Method:

It is the simplest method of hashing an Integer or data element. The hash function depends on the remainder of division.

The hash function is

h(key) = data % tablesize

Ex: if data 34, 57, 42, 69, 70 is to be placed in hash table of size 10 then

h(key) of 34 = 34 % 10 = 4

h(key) of 57 = 57 % 10 = 7

h(key) of 42 = 42 % 10 = 2

h(key) of 69 = 69 % 10 = 9

h(key) of 70 = 70 % 10 = 0

0	70
1	
2	42
3	
4	34
5	
6	
7	57
8	
9	69

Hash Table.

Mid-Square Method:

The mid-square method is a good hash function which works in two steps:

Step 1: Square the value of the key. That is, find k^2.

Step 2: Extract the middle r digits of the result obtained in Step 1.

1. Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations.

501:- $k = 1234$

$k^2 = 1234 * 1234 = 1522756$.

The hash value for keys are

$h(k) = 27$.

$k = 5642$

$k^2 = 5642 * 5642 = 31832164$

$h(5642) = 32$.

Multiplication Method:

The steps involved in the multiplication method are as follows:

Step 1: Choose a Constant A such that $0 < A < 1$

Step 2: Multiply the key K by A

Step 3: Extract the fractional part of KA

Step 4: Multiply the result of step 3 by the size of hash table (m).

Hence the hash function

$$h(K) = \lfloor m (KA \bmod 1) \rfloor$$

The Knuth morris has suggested that the best choice of A is

0.6180339887.

1. Given hash table size 1000 map the key 12345 to an appropriate location in the hash table.

$h(12345) = \lfloor 1000 * (0.6180339887 * 12345 \bmod 1) \rfloor$

$= \lfloor 1000 * 0.618\overset{7835}{\cancel{0339887}} \rfloor$

617.385.

617.

Folding Method: or Digit Folding Method:

The folding method works in the following two steps:

Step 1: Divide the key into number of parts. i.e., divide K into parts $K_1, K_2, \ldots K_n$ where each part has the same number of digits except the last part which may have lesser digits than the other parts.

Step 2: Add individual parts. That is Obtained the sum of $K_1 + K_2 + K_3 + \ldots + K_n$. The hash value is produced by ignoring the last carry, if any.

1. Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567.

	5678	321	34567
parts	56 ; 78	32, 1	34, 56, 7
Sum	1134	33	97
Hash value	34	33	97

Characteristics of Good Hashing Function:

1) The hash function should be simple to compute.
2) Number of collision should be less. Ideally no collision should occur. Such a function is called perfect hash function
3) Hash functions should produce such a keys which will get distributed uniformly over an array.
4) Hash function should depend upon every bit of the key. i.e, the hash function that simply extracts the portion of a key is not suitable.

Collisions:

Collisions occur when the hash function maps two different keys to the same location. Obviously, two records cannot be stored in the same location. Therefore a method used to solve the problem of collision, is called collision resolution technique.

The two most popular methods of resolving collisions are:

Open Addressing

Chaining.

Collision Resolution by Open Addressing:

Once a collision occurs, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position. In this technique all the values are stored in the hash table.

The hash table contains two types of values: Sentinel values (e.g. -1) and data values.

The presence of a sentinel value indicates that the location contains no data value at present but can be used to hold a value.

When a key or element is mapped to a particular memory location, then the value it holds is checked. If it contains a sentinel value, then the location is free and the data value can be stored in it. If the location already has some data value stored in it, then other slots are examined systematically in the forward direction to find a free slot. If even a single free location is not found, then we have an overflow condition.

The process of examining memory locations in the hash table is called probing. Open Addressing technique can be implemented using linear probing, quadratic probing, double hashing and rehashing.

Linear Probing:

The simplest approach to resolve a collision is Linear Probing. In this technique, if a value is already stored at a location generated by $h(k)$, then the following hash function is used to resolve the collision:

$$h(k, i) = [h'(k) + i] \bmod m$$

where m is the size of the hash table.

i is the probe number varies from 0 to $m-1$.

$$h'(k) = k \bmod m.$$

Ex: Consider a hash table of size 10. Using Linear probing insert the keys 72, 77, 36, 24, 63, 81, 92 and 101 into the table.

Sol: Let $h'(k) = k \bmod m$, $m = 10$

Initially the hash table can be given as

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Step 1

Key = 72

$h(72, 0) = (72 \bmod 10) \bmod 10$

$= 2 \bmod 10 = 2.$

Since T[2] is -1 insert 72 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step 2 Key = 27

$h(27, 0) = (27 \bmod 10 + 0) \bmod 10$

$= 7 \bmod 10$

$= 7$

Since T[7] is vacant insert Key 27 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Step 3 Key = 36

$h(36, 0) = (36 \bmod 10 + 0) \bmod 10$

$= 6 \bmod 10$

$= 6$

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Step 4

Key = 24

$h(24, 0) = (24 \bmod 10 + 0) \bmod 10$

$= 4 \bmod 10$

$= 4$

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Step 5 Key = 63

$h(63, 0) = (63 \bmod 10 + 0) \bmod 10$

$= 3 \bmod 10$

$= 3$

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Step 6 Key = 81

$h(81, 0) = (81 \bmod 10 + 0) \bmod 10$

$= 1 \bmod 10$

$= 1$

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Step 7:

Key = 92

$h(92, 0) = (92 \bmod 10 + 0) \bmod 10$

$= 2 \bmod 10$

$= 2$

Now T[2] is occupied so we cannot store the key 92 in T[2]. Therefore, try again for the next location. Thus probe i=1, this time

Key = 92

$h(92, 1) = (92 \bmod 10 + 1) \bmod 10$

$= (2+1) \bmod 10$

$= 3 \bmod 10$

$h(92, 1) = 3$

Now T[3] is occupied so we cannot store the key 92. Therefore try again for the next location. i=2

Key = 92

$h(92, 2) = (92 \bmod 10 + 2) \bmod 10$

$= 4 \bmod 10$

Now T[4] is occupied, so we cannot store the key 92 in T[4]. Therefore try again

again for the next location.

$i = 3$

Key = 92

$h(92, 3) = (92 \bmod 10 + 3) \bmod 10$

$= (2+3) \bmod 10$

$= 5$

Since T[5] is vacant insert key 92 at this location.

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	92	36	27	-1	-1

Quadratic Probing:

In this technique, if a value is already stored at a location generated by $h(k)$, then the following hash functions are used to resolve the collision.

$$h(k, i) = [h'(k) + c_1 i + c_2 i^2] \bmod m.$$

'm' is the size of hash table

'i' is the probe number varies from 0 to m-1.

$$h'(k) = k \bmod m.$$

C1, C2 are Constants such that $c_1 \neq 0$ and $c_2 \neq 0$.

(or)

$$h(k, i) = (h(k) + i^2) \bmod m.$$

'm' is the size of hash table

'i' is the probe number varies from 0 to m-1.

Quadratic probing eliminates the primary clustering phenomenon of Linear probing because

Instead of doing a Linear Search, it does a quadratic Search.

Consider the elements 37, 90, 55, 22, 17, 49, 87, to insert into a hash table of size 10.

Now Consider the keys

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

The hash key of above keys or elements are as follows

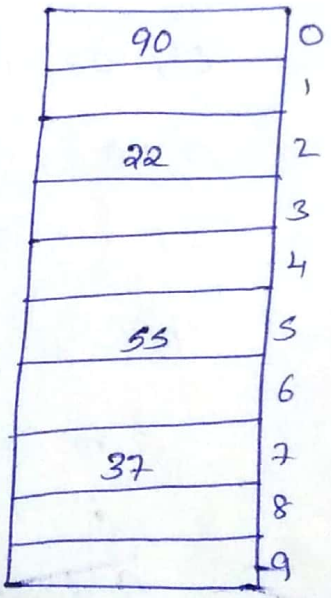
h(k) of 37 = 37 % 10 = 7

h(k) of 90 = 90 % 10 = 0

h(k) of 55 = 55 % 10 = 5

h(k) of 22 = 22 % 10 = 2

h(k) of 17 = 17 % 10 = 7 already occupied.



Here '7' positions already had a element 37. so we apply quadratic probing to insert 17.

$h(k, i) = (h'(k) + i^2) \bmod m$

$i = 0$

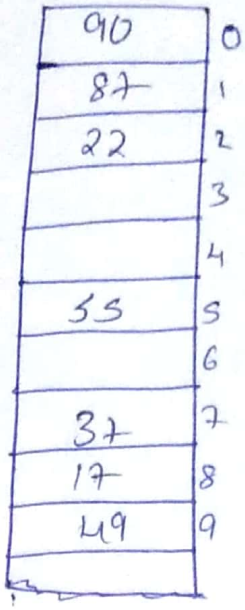
$h(k, 0) = (17 \% 10 + 0) \bmod 10$

$= 7$ already filled.

$i = 1$

$h(k, 1) = (17 \% 10 + 1^2) \bmod 10$

$= 18 \bmod 10 = 8.$



The 8th position in the hash table is empty so 17 can be placed at index 8.

$h(k)$ of $49 = 49 \% 10 = 9$

$h(k)$ of $87 = 87 \% 10 = 7$

7th position is already occupied apply quadratic probing

$h(k, i) = (h'(k) + i^2) \bmod m$

$= (87 + 0) \bmod 10 = 7$

$h(k, 1) = 88 \bmod 10 = 8$

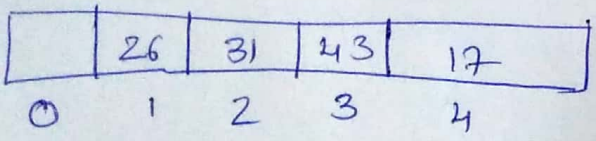
$h(k, 2) = (87 + 4) \bmod 10 = 91 \bmod 10 = 1$

Rehashing :-

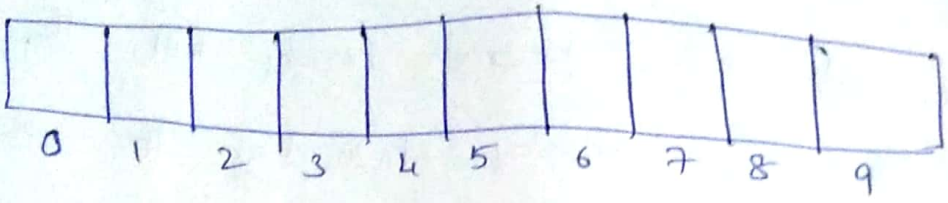
When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and deletion search operations. In such cases, a better option is to create a new hash table with size double of the original hash table.

All the entries in the original hash table will then have to be moved to the new hash table. This is done by taking each entry, computing its new hash value, and then inserting it in the new hash table.

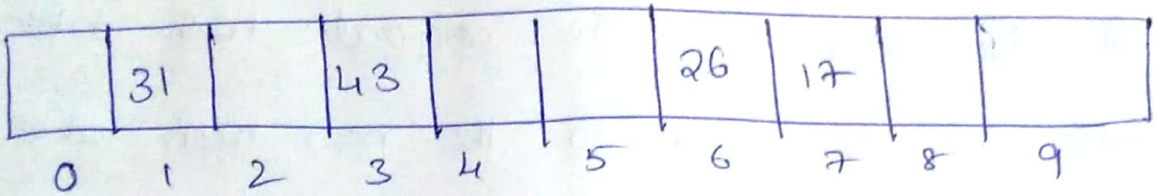
Though rehashing seems to be a simple process, it is quite expensive and must therefore not be done frequently. Consider the hash table of size 5 given below. The hash function used is $h(x) = x \% 5$. Rehash entries into a new ~~to~~ hash table.



Note that the new hash table is of 10 locations, double the size of the original table.

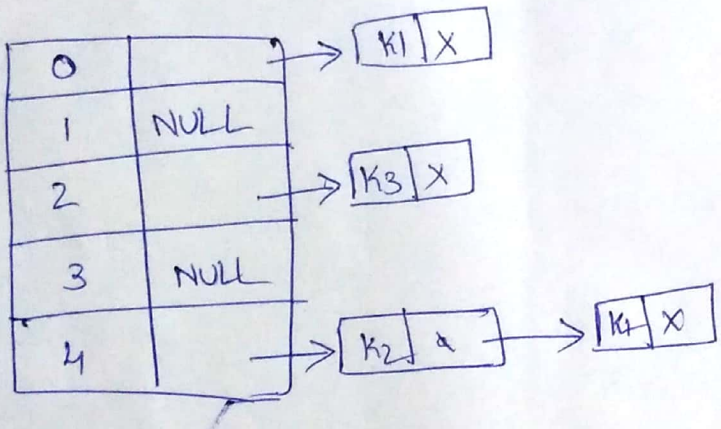


Now rehash the key values from the old has table into new one using hash function - $h(x) = x \% 10$



Collision Resolution by Chaining:-

In Chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location. i.e., location l in the hash table points to the head of the linked list of all the key values that hashed to l. However if no key value hashes to l, then location l in the hash table contains NULL.



Keys being hashed to a chained hash table.

Ex: Insert the keys 7, 24, 18, 52, 36, 54, 11 & 23 in a chained hash table of 9 mem locations. Use $h(k) = k \bmod m$.

Double Hashing:- It is a technique in which a second hash function is applied to the key when a collision occurs. By Applying the second hash function we will get the number of positions from the point of collision to insert.

- There are two important rules to be followed for the second function:
  - it must never evaluate to zero.
  - must make sure that all cells can be probed.

The formula to be used for double hashing is

$$H_1(key) = key \bmod tablesize.$$

$$H_2(key) = M - (key \bmod M)$$

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

Ex:- 37, 90, 45, 22, 17, 49, 55 — table size = 10.

Initially insert the elements using the formula for $H_1(key)$

Insert 37, 90, 45, 22

$H_1(37) = 37\%10 = 7$

$H_1(90) = 90\%10 = 0$

$H_1(45) = 45\%10 = 5$

$H_1(22) = 22\%10 = 2$

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

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

Now insert 17 is to be inserted then

$H_1(17) = 17\%10 = 7$ Collition occcured.

$H_2(key) = M1 - (key \% M1)$

Here M1 is a prime number Smaller than the size of the table.

prime number Smaller than table size 10 is 7.

Hence M1 = 7

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

$= 7 - 3$

$= 4$

that means we have to insert the element 17 at 4 places from 37.

In short we have to take 4 jumps. Therefore the 17 will be placed at index 1.

Now insert number 55.

$H_1(55) = 55\%10 = 5$ - Collition occcured.

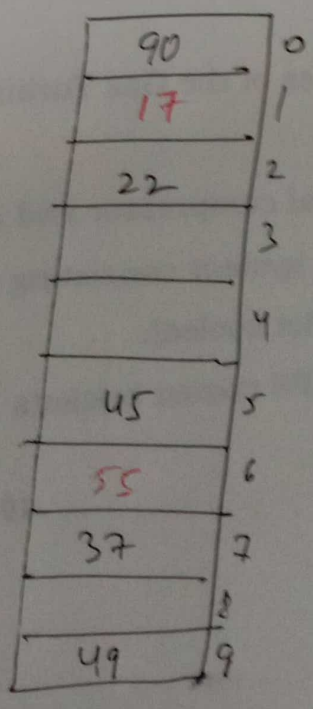
$H_2(55) = 7 - (55\%7)$

$= 7 - 6$

$= 1.$

That means we have to take one jump from index 5 to place 55.

Finally hash table is.



Double Hashing:- It is a technique in which a second hash function is applied to the key when a collision occurs. By Applying the second hash function we will get the number of positions from the point of collision to insert.

- There are two important rules to be followed for the second function:
  - it must never evaluate to zero.
  - must make sure that all cells can be probed.

The formula to be used for double hashing is

$$H_1(key) = key \bmod tablesize.$$

$$H_2(key) = M - (key \bmod M)$$

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

Ex:- 37, 90, 45, 22, 17, 49, 55 — table size = 10.

Initially insert the elements using the formula for $H_1(key)$

Insert 37, 90, 45, 22

$H_1(37) = 37\%10 = 7$

$H_1(90) = 90\%10 = 0$

$H_1(45) = 45\%10 = 5$

$H_1(22) = 22\%10 = 2$

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

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

Now insert 17 is to be inserted then

$H_1(17) = 17\%10 = 7$ Collition occcured.

$H_2(key) = M1 - (key \% M1)$

Here M1 is a prime number Smaller than the size of the table.

prime number Smaller than table size 10 is 7.

Hence M1 = 7

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

$= 7 - 3$

$= 4$

that means we have to insert the element 17 at 4 places from 37.

In short we have to take 4 jumps. Therefore the 17 will be placed at index 1.

Now insert number 55.

$H_1(55) = 55\%10 = 5$ - Collition occcured.

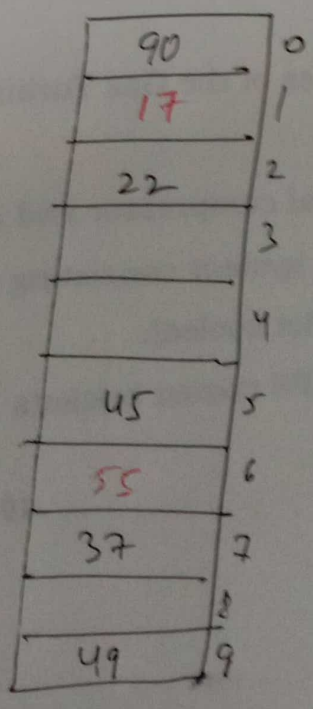
$H_2(55) = 7 - (55\%7)$

$= 7 - 6$

$= 1.$

That means we have to take one jump from index 5 to place 55.

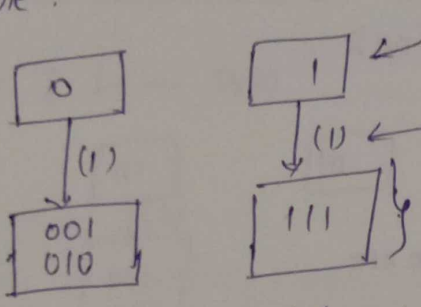
Finally hash table is.



Extensible Hashing:- It 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.

- Extensible hashing grow and shrink similar to B-trees.
- In extensible hashing referring the size of directory the elements are to be placed in buckets. The levels are indicated in paranthesis.

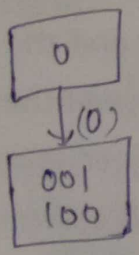
for Example:



- 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.

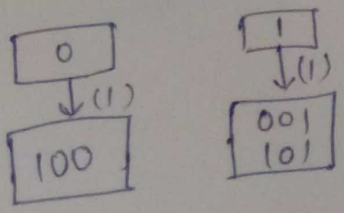
- Ex:- Consider we have to insert 1, 4, 5, 7, 8, 10 Assume each page can hold 2 data entries (2 is the depth.)

Step 1: Insert 1, 4



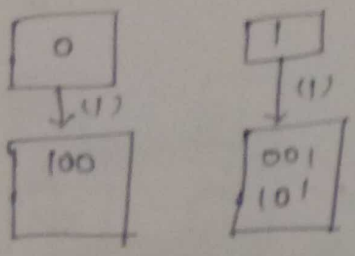
1 = 001
4 = 100
we will examine last bit of data and insert the data in bucket.

insert 5. The bucket is full. Hence double the directory



1 = 001
4 = 100
5 = 101
Based on last bit the data is inserted.

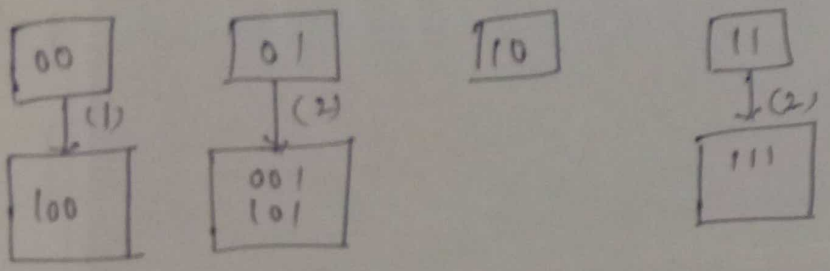
Step 2: Insert 7



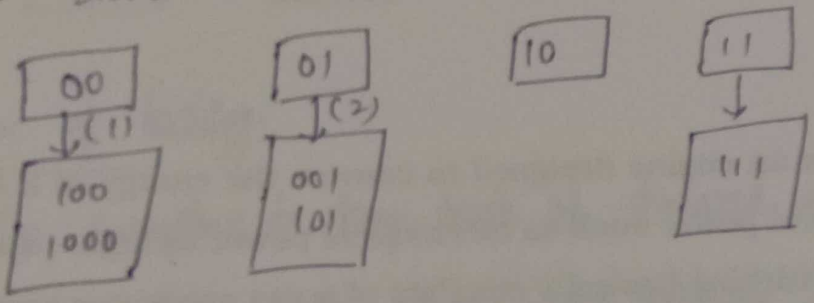
7 = 111

Insert 7: But as depth is full we can not insert 7 here. Then double the directory and split the bucket.

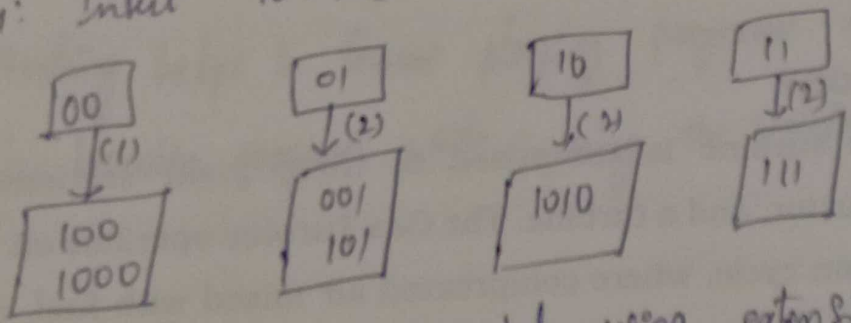
After insertion of 7. Now consider last two bits.



Step 3: Insert 8 i.e — 1000



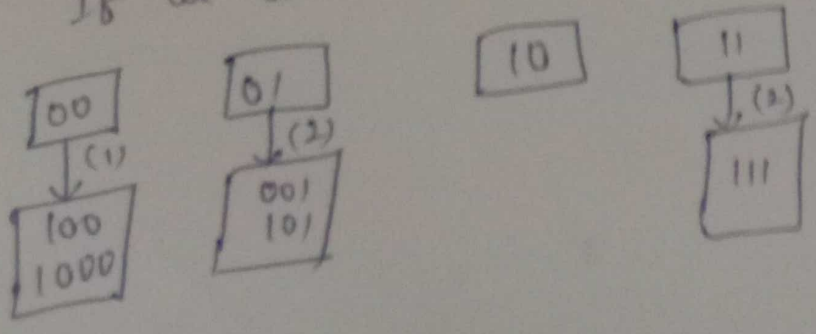
Step 4: Insert 10. i.e 1010.



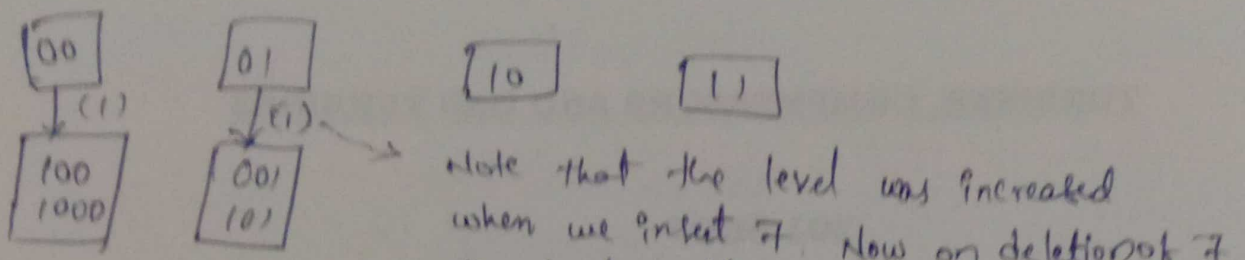
Thus the data is inserted using extensible hashing.

Deletion operation

If we want to delete 10 then, simply make the bucket of 10 empty.

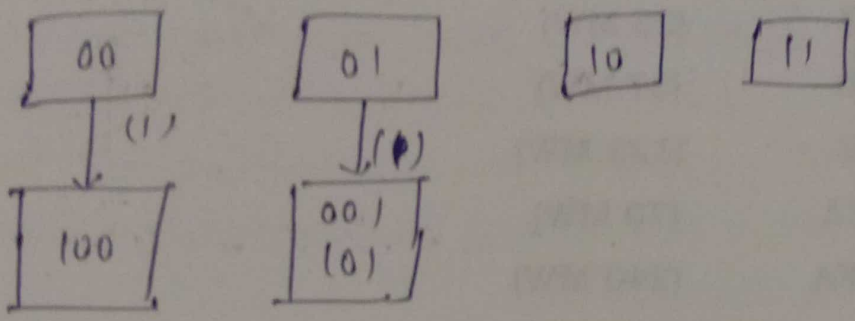


Delete 7:



Note that the level was increased when we insert 7. Now on deletion of 7, the level should be decremented.

Delete 8: Remove entry from directory 00.



Applications of hashing:

- In compilers to keep track of declared variables.
- For online spelling checking the hashing functions are used.
- Hashing helps in Game playing programs to store the moves made.
- For browser program while caching the webpages, hashing is used.