UNIT-II

Dictionaries:

Def:- Dictionary is a collection of ~~Pairs of~~ key value pairs. Every ~~key~~ value is associated with the corresponding key.

Advantages of dictionaries:

① Speed: By using dictionaries increase the process speed for some type of data.

② Memory Efficient:- By using dictionaries we can efficiently store ~~the~~ the data for tries and hash tables.

③ Flexibility - Easily extend and modify the existing dictionaries.

④ Unordered - ✳

Disadvantages of dictionaries:-

① Complexity:- To implement dictionaries little bit Complex.

② Space Complexity:- Depending on the size of the dictionary and choosen data structure, the memory requirement ~~is~~ can be quite high.

Example for dictionary.

Following example key is the rollnumber and his exam score as value.

Key value pair is {rollno: score}
key value.

{545 : 80, 580 : 85, 510 : 80}

In key value pair key must be unique.

If key is already exited it can be update with new value.

If key is not exited it can be added as new dictionary element.

Dictionary representation:-

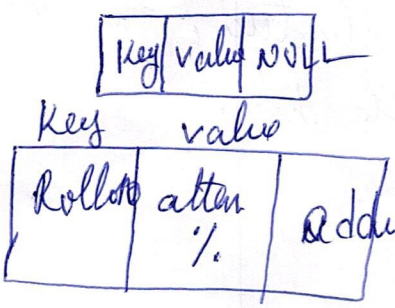
There are two types of representations

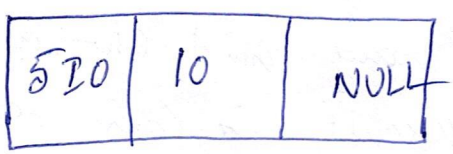
① Linear list representation.

② Skip list representation.

Linear List representation:-

It is almost same as linked list, here every node maintain the key value pair.






```
typedef struct node dnode;

struct node
{
    int key;
    int value;
    struct node *next;
};
```

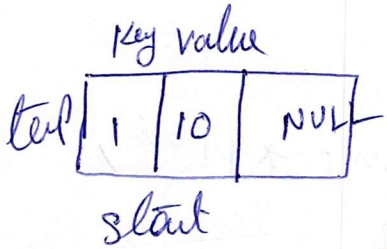
	Insertion	Removal	Retrieval
unsorted array	O(1)	O(n)	O(n)
unsorted list	O(1)	O(n)	O(n)
sorted array	O(n)	O(n)	O(log n)
sorted list	O(n)	O(n)	O(n).

Operations on dictionaries:-

1. Insert / update
2. Delete
3. Retrieve / search
4. Traverse / display.

Insert:- Consider initially the dictionary is empty, that means start is NULL.

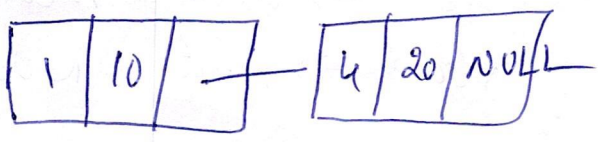
Create a new node with some key value.



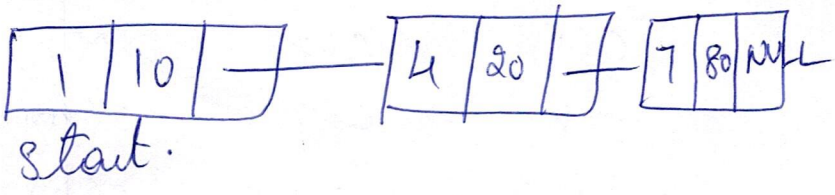
now temp node becomes start.

If dictionary is not empty find appropriate position then insert.

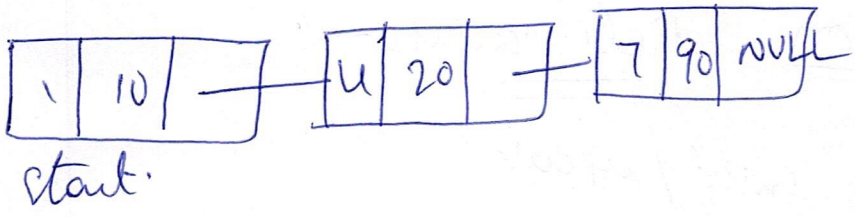
now insert (4, 20)



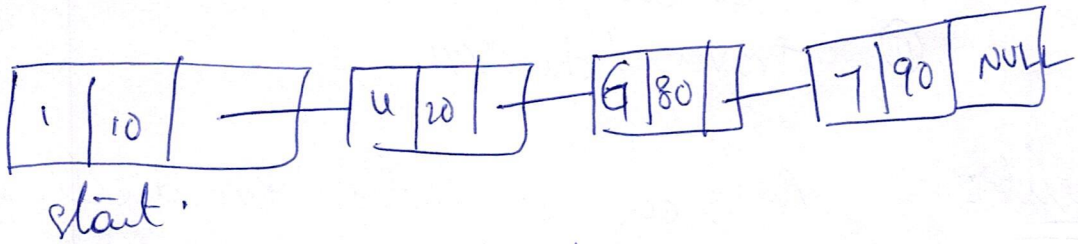
now insert (7, 80)



now insert (7, 90)



now insert (6, 80)



algorithm insert (key, value)

{

// start is global identifier represents the head node.

dnode *temp, *cur, *prev;

cur = prev = start;

if (start == NULL)

{

start = malloc (sizeof (dnode));

start -> key = key;

start -> value = value;

start -> next = NULL;

}

else {

```
cur = prev = start
while((cur->key != key) && (cur->next != NULL)&&
                          (cur->key < key))
    {
        prev = cur;
        cur = cur->next;
    }
// update with existing key
if ( cur->key == key)
    cur->value = value;
else
    if ( cur->key > key) // insert at middle position.
        {  temp= malloc (sizeof(node));
           temp->key = key; temp->val = value;
           prev->next = temp;
           temp->next = cur;
        }
    else
        if ( cur == start && cur->key > key)
            {
                // create node temp;
                temp->next = start;
                start = temp;
            }
        else
            {
                cur->next = temp;
                temp->next = NULL;
            }
    }
}
```

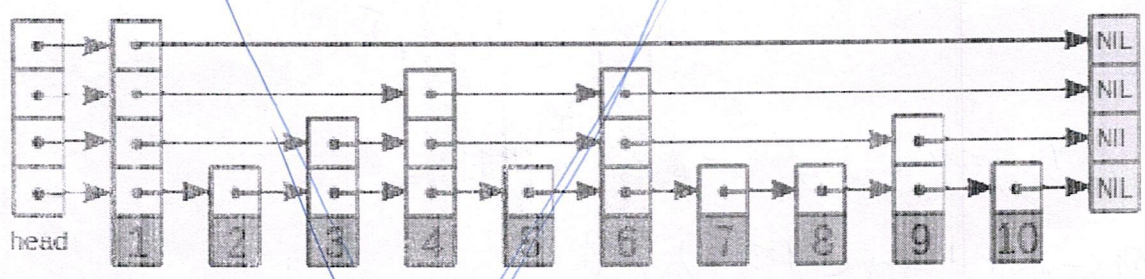
```
delete (key)
{
// start is global variable

cur = prev = start;
if (cur == NULL)
    Put(" dictionary is empty ");
else
    {
        while (cur->key != key && cur->next != NULL)
            {
                prev = cur;
                cur = cur->next;
            }
        if (cur == start)
            {
                start = start->next;
                free (cur);
            }
        else if (cur->next == NULL && cur->key != key)
                printf ("elemt not found");
            else
                if (cur->key == key) && (cur->next == NULL)

                    {
                        prev->next = NULL;
                        free (prev);
                    }
                else
                    {
                        prev->next = cur->next;
                        free (cur);
                    }
}
```

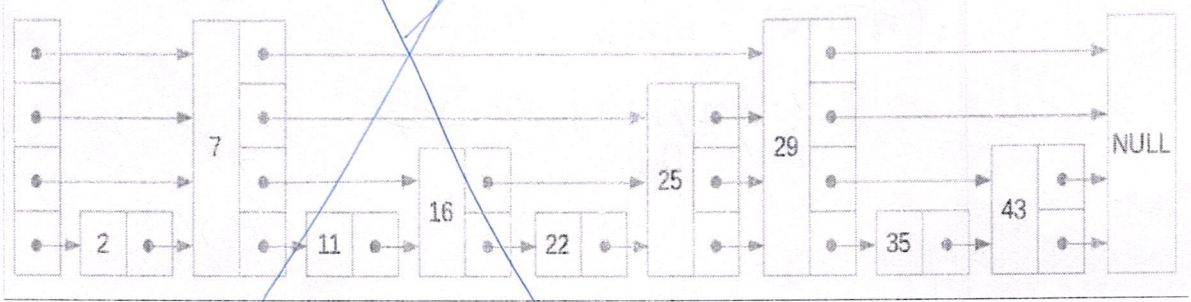
```
Traverse/display()
{
  // start is global identifier to represent
     start node of the dictionary.
  if (start == NULL)
    {
       Put ("Dictionary is empty");
    }
  else
    {
       start temp = start;
       while(temp != NULL).
         {
           Put ("temp -> key, temp -> value);
           temp = temp -> next;
         }
    }
}
search/Retrieve (key)
{
  // start - -
  if (start == NULL)
     Put("dictionary is empty");
  else
    {
       curr = prev = start;
       while (curr -> next != NULL && curr -> key != key)
         {
           prev = curr;
         } curr = ~~-> next~~ curr -> next;
```

Skip List

A skip list is a data structure that is used for storing a sorted list of items with a help of hierarchy of linked lists that connect increasingly sparse sub sequences of the items.

A skip list allows the process of item look up in efficient manner. The skip list data structure skips over many of the items of the full list in one step, that's why it is known as skip list.





Structure of Skip List

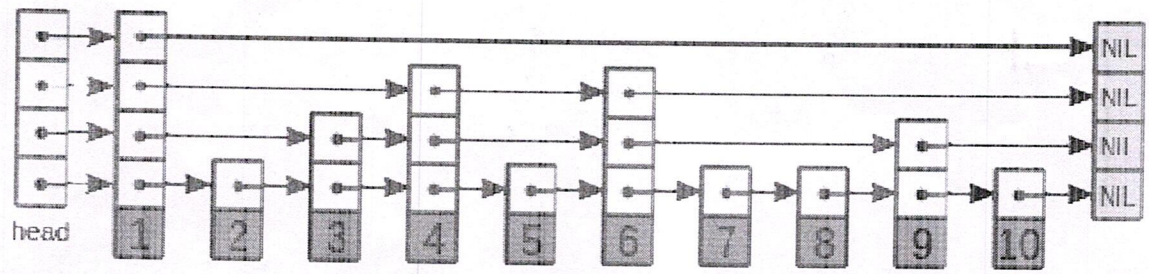
A skip list is built up of layers. The lowest layer (i.e. bottom layer) is an ordinary ordered linked list. The higher layers are like 'express lane' where the nodes are skipped (observe the above figure).

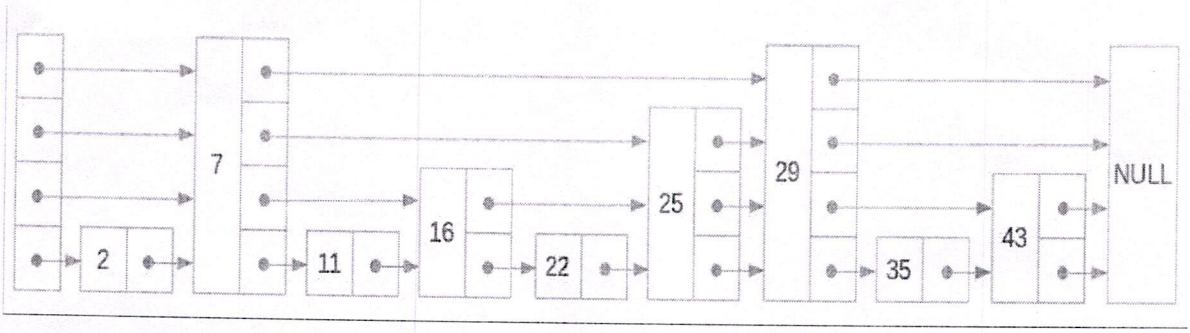
```
if (cur -> key = key)
    Put("key found and its value is ", cur -> value);
else
    Put("key not found");
```

Skip List

A skip list is a data structure that is used for storing a sorted list of items with a help of hierarchy of linked lists that connect increasingly sparse sub sequences of the items.

A skip list allows the process of item look up in efficient manner. The skip list data structure skips over many of the items of the full list in one step, that's why it is known as skip list.





Structure of Skip List

A skip list is built up of layers. The lowest layer (i.e. bottom layer) is an ordinary ordered linked list. The higher layers are like 'express lane' where the nodes are skipped (observe the above figure).

Advantages of the Skip list

1. The skip list is simple to implement as compared to the hash table and the binary search tree.
2. It is very simple to find a node in the list because it stores the nodes in sorted form.
3. The skip list is a robust and reliable list.

Disadvantages of the Skip list

1. It requires more memory than the balanced tree.
2. Reverse searching is not allowed.
3. The skip list searches the node much slower than the linked list.

Applications of the Skip list

1. It is used in distributed applications.
2. It is also used with the QMap (Open Multimedia Application Platform) template class.
3. The indexing of the skip list is used in running median problems.

Searching Process

When an element is tried to search, the search begins at the head element of the top list. It proceeds horizontally until the current element is greater than or equal to the target. If current element and target are matched, it means they are equal and search gets finished.

Skip List Basic Operations

There are the following types of operations in the skip list.

- **Insertion operation:** It is used to add a new node to a particular location in a specific situation.
- **Deletion operation:** It is used to delete a node in a specific situation.
- **Search Operation:** The search operation is used to search a particular node in a skip list.

Insertion operation:

Example 1: Create a skip list, we want to insert these following keys in the empty skip list.

1. 6 with level 1.
2. 9 with level 3.

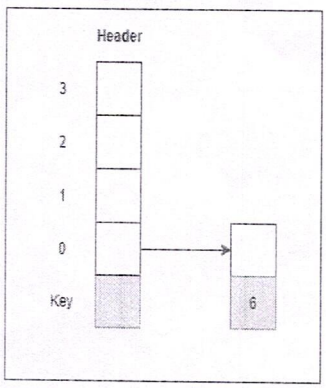
29 with level 1.

17 with level 1.

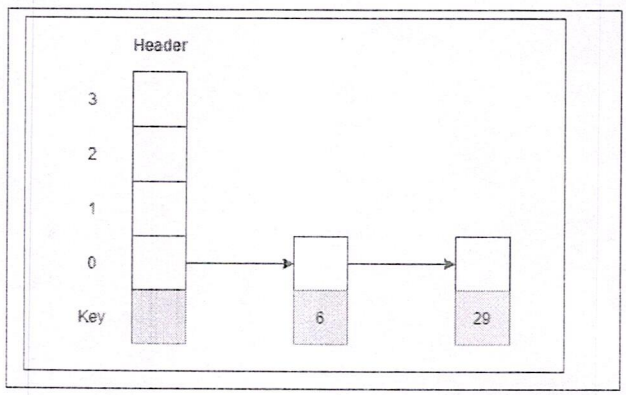
22 with level 4.

4 with level 2.

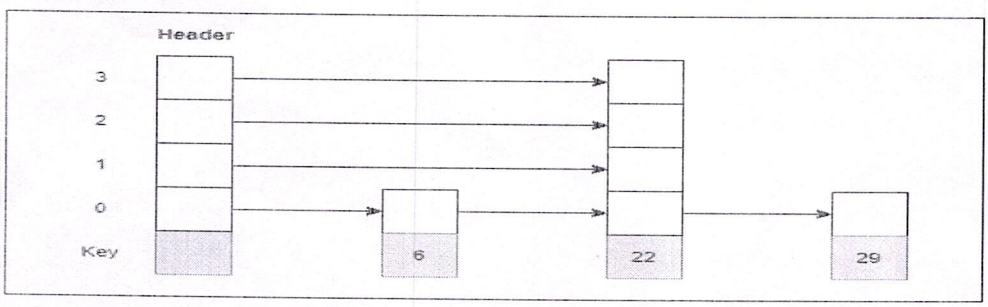
6 with level 1.



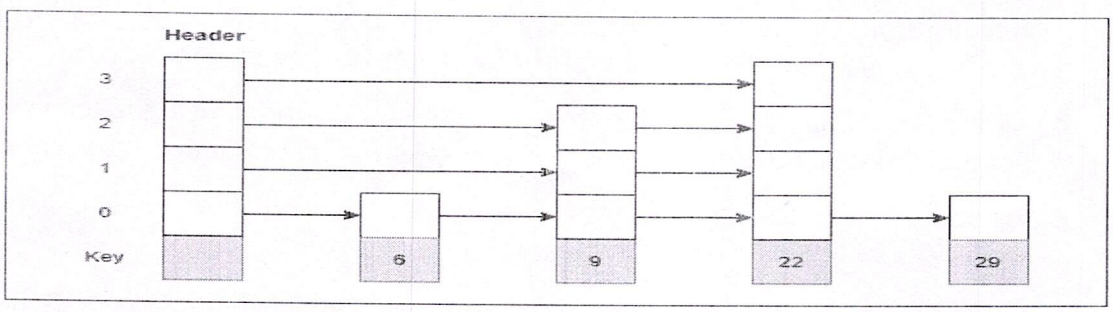
29 with level 1.



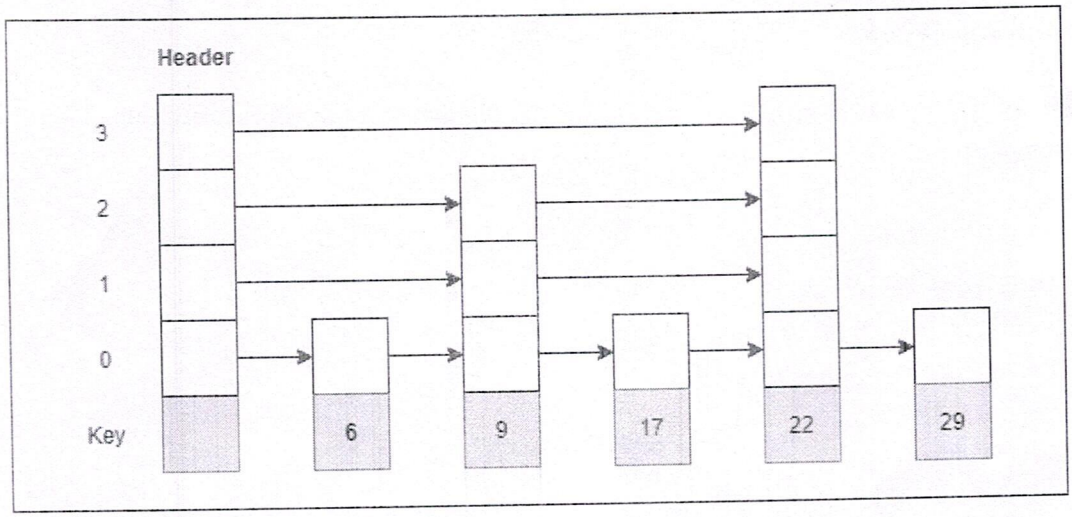
Insert 22 with level 4



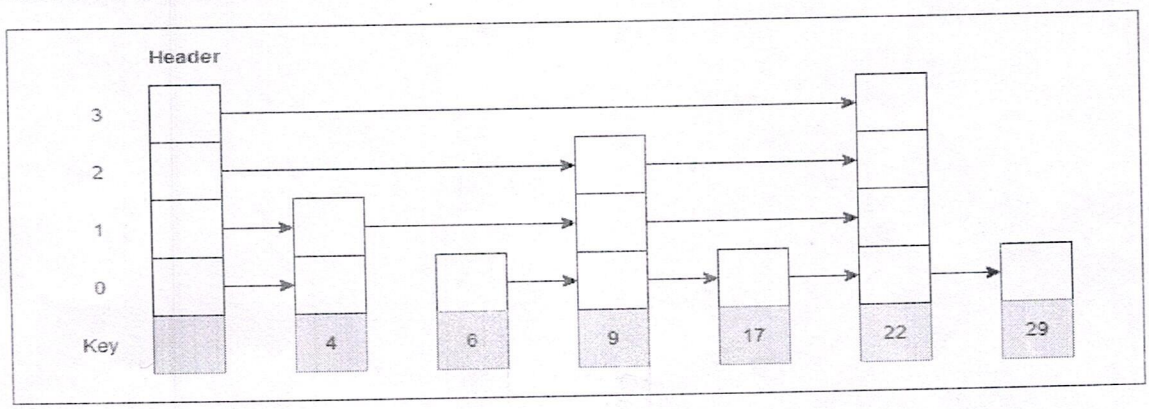
9 with level 3.



Insert 17 with level 1



Insert 4 with level 2



Algorithm steps to insert element into skip list.

1. Create a new node with the value you want to insert.
2. Determine the level of the new node. You can do this randomly by flipping coins or using some other probabilistic method. The higher the node's level, the fewer nodes it will skip, leading to faster access times. Typically, the maximum level of a skip list is around log2(n), where n is the number of elements in the list.
3. Start at the top-left corner (head) of the skip list and traverse the list horizontally while moving to the right until you find a node with a smaller value or reach the end of the list at the current level.
4. When you find the right spot, insert the new node into the current level's linked list by updating the pointers of the adjacent nodes.
5. Repeat this process for each level, going down one level at a time until you reach the base level.
6. The insertion is complete, and the new node is successfully added to the skip list

Algorithm steps to delete element from skip list.

1. Start at the top-left corner (head) of the skip list and traverse the list horizontally while moving to the right until you find the node with the target value or reach the end of the list at the current level.
2. When you find the node with the target value, you need to update the pointers of the adjacent nodes to remove the node from the skip list. To do this, you'll need to keep track of the update nodes (similar to the insertion algorithm).
3. Once you've updated the pointers at the current level, repeat this process for each level, going down one level at a time, until you reach the base level.
4. After completing the deletion for all levels, the node with the target value is successfully removed from the skip list.

Algorithm steps to delete element from skip list.

1. Start at the top-left corner (head) of the skip list and traverse the list horizontally while moving to the right until you find the node with the target value or reach the end of the list at the current level.
2. When you find the node with the target value, you need to update the pointers of the adjacent nodes to remove the node from the skip list. To do this, you'll need to keep track of the update nodes (similar to the insertion algorithm).
3. Once you've updated the pointers at the current level, repeat this process for each level, going down one level at a time, until you reach the base level.
4. After completing the deletion for all levels, the node with the target value is successfully removed from the skip list.

Hashing

Hash :- Hashing is technique to map certain data to a particular key. In a large database, the data can be quickly stored and retrieving using the key.

Hash Table :- We store the identifiers in a fixed size table called hash table.. we use the arithmetic function f to determine the address or location.

So $f(x)$ gives the hash or home address of x in the table.

Hash functions :-

A hash function f transforms an identifier x into a bucket address in the hash table. We need a hash function that is easy to compute and that minimizes the number of collisions.

Types of hash functions :- There are diffrent of hash functions to find/calculate the hash bucket address.

1. Division method
2. multiplication method
3. Mid Square method
4. Digit folding method
5. Digit analysis method.

Division method :- This method used to modulo operator to compute the hash code for a key. The modulo operator returns the remainder of the key.

Example : size of the array is - 10

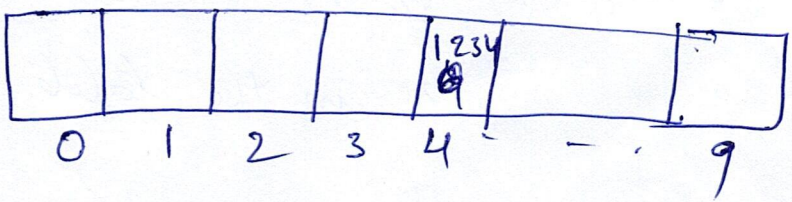
and key is 1234

division method function

$f(\text{key}) = \text{key} \% \text{size}$.

$f(1234) = 1234 \% 10$

$= 4$



Note :- size of the array take it as prime number so it can reduce the number of collisions.

Advantage :- It gives the fast access.

Multiplication method :-

This method involves the key by a constant value then taking fractional part of the result. The fractional part is multiplied by the size of the table, and then take floor of that value taken as the hash code.

$h(key) = floor(size(KA \bmod 1))$

K is the key

A is constant — 0.6180339887

size is size of the table — 13

$h(123) = floor(13(123 * 0.6180339887 \% 1))$

$= floor(13 * (0.01818))$

$= floor(0.236348)$

$= 0$

$h(125) = floor(13(125 * 0.6180339887 \% 1))$

$= floor(13(77.254625 \% 1))$

$= floor(13(0.254249))$

$= floor(3.305232)$

$= 3$

Advantages: ① It produces the good distribution of keys.

② It is suitable for both real and integer keys.

Disadvantages:- ① It is more complex to implement.

② it may not work for small table size.

③

mid square method :-

This method involves squaring the key and extracting some digits from middle of the squared value.

$h(key) = mid(K * K)$;

$h(123) = mid(23 * 123)$

$= mid(15129)$

$= 1$

$h(56) = mid(56 * 56)$

$= mid(3136)$

$= 1$

$h(158) = mid(158 * 158)$

$= mid(24964)$

$= 9$

folding method :-

The key value K should be devided into a specific number of parts such as $K1, K2, \ldots Kn$. Add each component separately.

$K = K1, K2, \ldots Kn$.

$h(K) = K1 + K2 + K3 + \ldots + Kn$.

$h(K) = 8$.

Example:- K = 1 8 5 4 3 2 1

K1 = 54 K2 = 32 K3 = 1

S = K1 + K2 + K3

= 54 + 32 + 1

= 87

h(k) = 87.

advantage:- Independent distribution in hash table

disadvantage:-

It produces more collisions.

Collision resolution Techniques:-

what is collision:- Since hash function gets us a small number of a key which is a big integer or string, there is 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.

To handle this situation by using collision resolution methods.

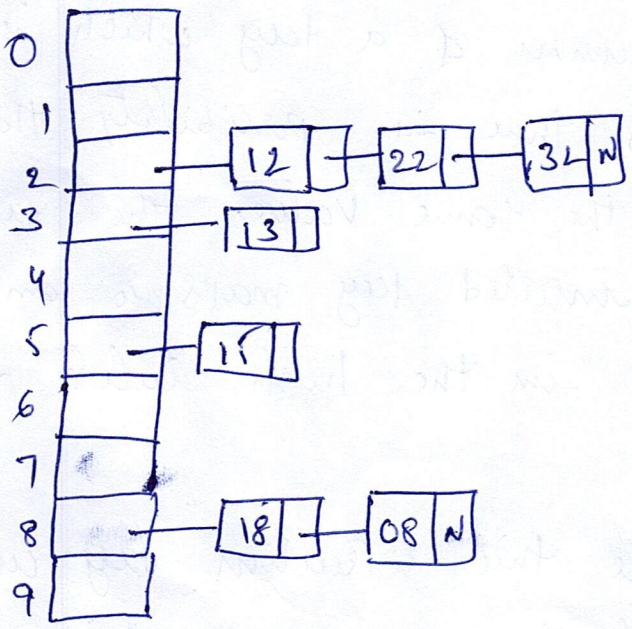
There are different types of collision resolution techniques

① Separate chaining
② open addressing
③ a) Linear probing
④ b) quadratic probing
⑤ c) double hashing
⑥ · Rehashing
⑦ Extensible hashing.

Separate Chaining:-

Separate Chaining is the collision resolution technique that is implemented using linked list. When two or more elements are hash to the same location, these elements are represented into a single linked list, like a chain. It is also known as open hashing.

12, 13, 15 18, 08, 22, 32



h(key) = key % 10

Advantages:- ① It is one of the simplest method to implement.

② We can add any number of element to the chain.

Disadvantages:- ① The keys in the hash table are not evenly distributed.

② Some amount of space wastage.

③ Complexity in searching is O(n) in worst case.

Open Addressing:- (closed hashing)

In open addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys. This approach is also known as closed hashing.

There are different ways to implement open addressing.

① Linear probing.

② Quadratic probing

③ Double hashing.

Linear probing:- Linear probing collisions are resolved by searching the hash table consequently until empty cell is found. This process is called linear probing.

NOTE: Probing:- Probing is the method in which to find an open bucket, or an element already stored in the underlaying array of a hash table.

the following formula to implent
By using a linear probing

$$f(x) = (h(x) + j) \bmod \text{size}.$$

f is the hash function
x is the value to be store.
j is iteration of the probe. (it values from 0 to K

Let us Consider the following data, store into the hash table of size 13 using linear probing.

0	1	2	3	4	5	6	7	8	9	10	11	12
	40	27	16	28					9	75		

hash table size is 13 then insert the following elements. 16, 40, 27, 9, 75, 28

insert - 16 → $f(16) = 16 \% 13 = 3$
insert - 40 → $f(40) = 40 \% 13 = 1$
insert - 27 → $f(27) \rightarrow 27 \% 13 = 1$

collision occur
So we need to probe for empty location
linear probing is

$j = 1$
$f(27) = f(27) + j$
$= 1 + 1 = 2$
insert 27 at 2nd position.

insert 9 → $h(9) = 9 \% 13 = 9$
insert 75 → $h(75) = 75 \% 13 = 10$

insert (28) = 28 % 13 = 2

Collision occured → j = 1

insert (28)$_1$ = (28 % 13) + 1

= 2 + 1

= 3

Third location also ~~inserted~~ contains other value so find next empty location.

j = 2

insert (28)$_2$ = ~~f~~(28)$_a$ + 2

= f(28) + j

= 2 + 2

= 4

fourth location is empty ht[4] = 28

search for 28 :-

find identify f(28) is 2

ht[2] ! = 28

find the probe

f(28) + 1 = 2 + 1 = 3

ht[3] ! = 28 go for next iteration.

f(28) + 2 = 2 + 2 = 4

ht[4] = 28 element not found.

note: if the location is empty simply you can say element not found.

Quadratic probing:- Quadratic probing is an open addressing method to resolve the collision in hash table. This method is eliminating the primary cluster problem which is created in linear probing.

This technique works by considering of original hash index and adding successive value of an ~~arbitrary~~ arbitrary quadratic polynomial until the empty location is found.

In linear probing we follow

$$f(x)+0,\ f(x)+1 \cdots f(x)+k.$$

In quadratic probing we follow

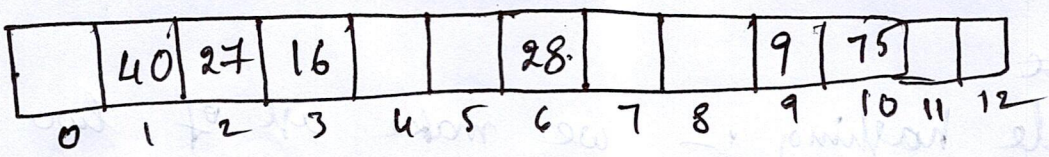
$$f(x)+0^2,\ f(x)+1^2,\ f(x)+2^2, \ldots$$
$$f(x)+k^2$$

So we can write the equation

$$H_i(x) = \left(f_i(x) + i^2\right) \% \text{.Sht}$$

Example:-

Let us consider the hash table size 13 then insert the following data 16, 40, 27, 9, 75, 28.



insert(16) = 16 % 13 = 3

insert(40) = 40 % 13 = 1

insert(27) = 27 % 13 = 1

Collision occured. Then apply quadratic probing.

$f(x) = (f(x) + i^2) \% \text{Sht}$.

$= ((27 \% 13) + i^2) \% 10$

$= (1 + 1^2) \% 10 = 2$

insert(9) = $((f(9) + 0^2)) \% 13$

$= (9 + 0) \% 13 = 9$

insert(75) = $(f(75) + 0^2) \% 13$

$= (10 + 0) \% 13$

$= 10 \% 13 = 10$

insert(28) = $(f(28) + 0^2) \% 13$

$= 2 \% 13 = 2$

Collision occured.

$(f(28) + 1^2) \% 13$

$= (2 + 1) \% 13$

$= 3$

still collision

$(f(28) + 2^2) \% 13$

$= (2 + 4) \% 13$

$= 6 \% 13 = 6$

~~Dynamic~~

Double hashing :- we make use of two hash functions. The first hash function is $h_1(k)$ ~~and~~ this function takes in our key and gives out a location on the hash table. If the location is empty easily insert the value.

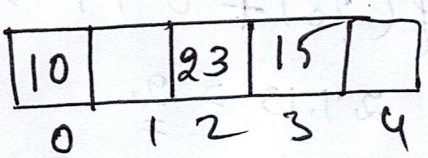
In case of collision, we need to use secondary hash function $h_2(k)$, in combination with $h_1(k)$.

The combined function is

$$f(k, i) = (h_1(k) + i * h_2(k)) \% \text{size}$$

i is non negative integer

Example lets consider the table of size 5 then insert 10, 15, 23



Consider $h_1(k)$ is $k \% 5$

$h_2(k)$ is $k \% 7$

insert (10) $= 10 \% 5 = 0$

insert (15) $= 10 \% 5 = 0$

Collision occured. $\rightarrow i = 1$

$= ((10 \% 5) + 1 * (10 \% 7)) \% 5$

$= (0 + 3) \% 5$

$= 3$

Insert (23) . 23 %. 5 = 3

Collision occured i = 1

= (23 %. 5 + 1 * 23 %7) %. 5

= (3 + 2) %. 5

= 5 %. 5

= 0

again occured the collision i = 2

= (23 %5 + 2 * 23 %7) %. 5

= (3 + 4) %. 5

= 7 %. 5 = 2

Rehashing:- Rehashing is the process of re-calculating the hash codes of already stored entries to move them to another bigger size hash table. when the threshold is reached/ crossed.

Rehashing will be implement hash table reaches the load factor of 0.75 or 75% of the hash table size.

Let us consider size is 17

so load factor is 0.75 * 17 = 12.75 = 13.

If the hash table occupancy is reached 13 then apply rehashing.

now the hash table size is ~~2*13~~ extend to next Prime number. or next primenumber (2 * size));

Extensible hashing :- Extendable hashing is a dynamically updatable disk-based index structure which implements a hashing technique utilizing a directory. The index is used to support exact match queries. i.e find the record with a given key compared with the B+ tree.

In this process maintain a directory of bucket address instead of ~~such~~ ~~hashing~~ hashing to buckets ~~to each~~ directory.

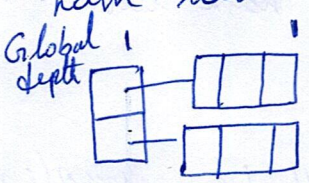
The directory can grow its size power of 2.

At any time, the directory consists of d levels

Let us consider directory size is 1

and Bucket size is 3

Now the hash table is



Global depth represents no. of MSB Most significant bits considered for hashing.

Let us Consider the example to insert the values into hash table using extendable hashing.

Consider the initial global depth is 1

and bucket size is 3

Values are 16, 5, 6, 22, 24, 11, 30, 7

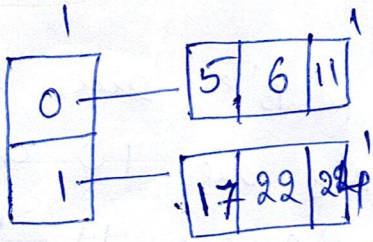
10, 21, 27.

first convert the value as binary value.

17	10001	30	11110
5	00101	7	00111
6	00110	10	01010
22	10111	21	10101
11	01011	27	11011
24	11000		

17, 5, 6, 22, 24 ~~##~~ 30 7 10 21 27

initially hash table is

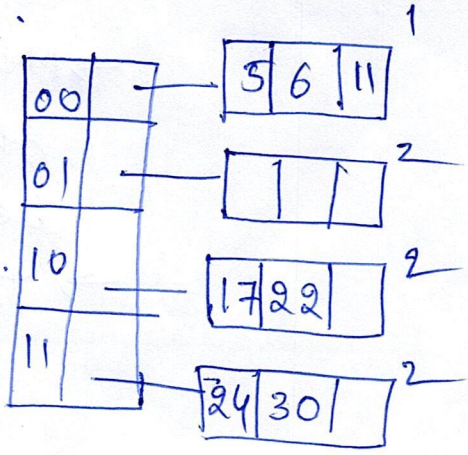


first insert 17
17 binary value is 10001
its MSB is 1 so insert at 1's point bucket.

insert 5 its bucket value is 0 bc2 of its MSB.
insert 6 its MSB also zero then insert at 0'
insert 22 its MSB is 1 insert after 17
insert 24 its MSB is 1 insert after 22.
insert 11 its MSB is 0 insert after 6
insert 30 its MSB is 1, But the corresponding bucket is full so increase the Directory size to power of ~~insert~~ next integer. (insert size is 1 it can be increase to 2)

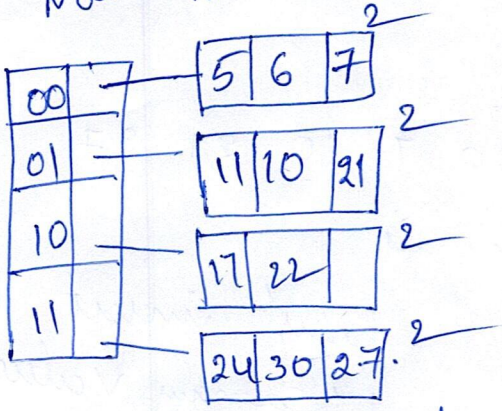
so power of 2 is 4.

Now the directory structure is like the next figure. Now you can split the overflow buckets. then apply rehashing. with 2 MSB Bits then the tablee.



Now insert 7 its MSB is 00 Bucket is already full with single MSB hash value. Then apply rehashing with 2 MSB's.

Now the table is



Now insert 10 its MSB's are 01 insert after 11
insert 21 its MSB's are 10 insert after 10.
insert 27 its MSB's are 11 insert after 30.

Construct the hash table with the following data.

28, 4, 19, 1, 22, 16, 12, 0, 5, 7.

28 - 11100 4 - 00100 19 - 10011 1 - 00001
22 - 10110 16 - 10000 12 - 01100 0 - 00000
05 - 00101 7 - 00111

Advantages :- → Data retrival is less expensive
→ Dynamic changes in hashing function.

Limitations :- → Size of every bucket is fixed.
→ Memory is wasted in pointers when the global depth and local depth difference becomes drastic.
→ This method is complicated.