UNIT-2

Dictionaries: linear list representation, skip list representation, operations - insertion, deletion and searching. Hash Table Representation: hash functions, collision resolution-separate chaining, open addressing-linear probing, quadratic probing, double hashing, rehashing, extendible hashing.

DICTIONARIES:

Dictionary is a collection of pairs of key and value where every value is associated with the corresponding key.

Basic operations that can be performed on dictionary are:

1. Insertion of value in the dictionary
2. Deletion of particular value from dictionary
3. Searching of a specific value with the help of key

Linear List Representation

The dictionary can be represented as a linear list. The linear list is a collection of pair and value. There are two method of representing linear list.

1. Sorted Array- An array data structure is used to implement the dictionary.
2. Sorted Chain- A linked list data structure is used to implement the dictionary

Structure of linear list for dictionary:

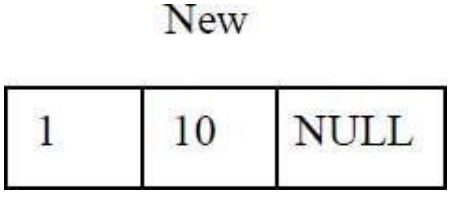
```
class dictionary
{

private:
        int k,data;
        struct node
        {
        public: int key;
        int value;
        struct node *next;
        } *head;



public:
        dictionary();
        void insert_d( );
        void delete_d( );
        void display_d( );
        void length();
};
```

Insertion of new node in the dictionary:

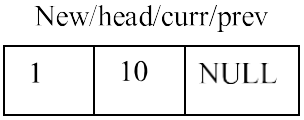
Consider that initially dictionary is empty then

head = NULL

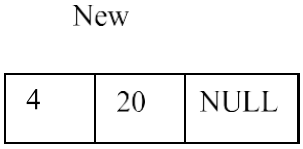
We will create a new node with some key and value contained in it.



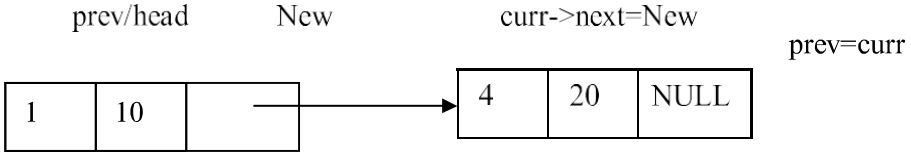
Now as head is NULL, this new node becomes head. Hence the dictionary contains only one record. this node will be 'curr' and 'prev' as well. The 'cuur' node will always point to current visiting node and 'prev' will always point to the node previous to 'curr' node. As now there is only one node in the list mark as 'curr' node as 'prev' node.



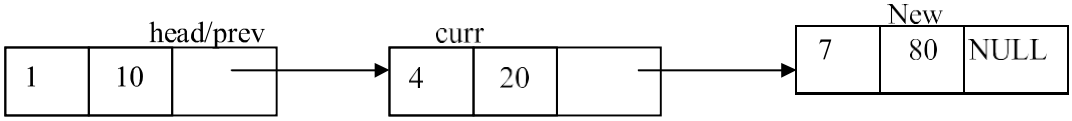
Insert a record, key=4 and value=20,



Compare the key value of 'curr' and 'New' node. If New->key > Curr->key then attach New node to 'curr' node.

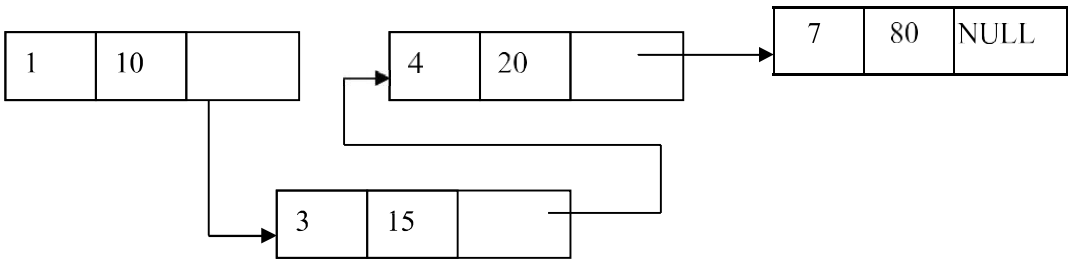


Add a new node <7,80> then



If we insert <3,15> then we have to search for it proper position by comparing key value.

(curr->key < New->key) is false. Hence else part will get executed.

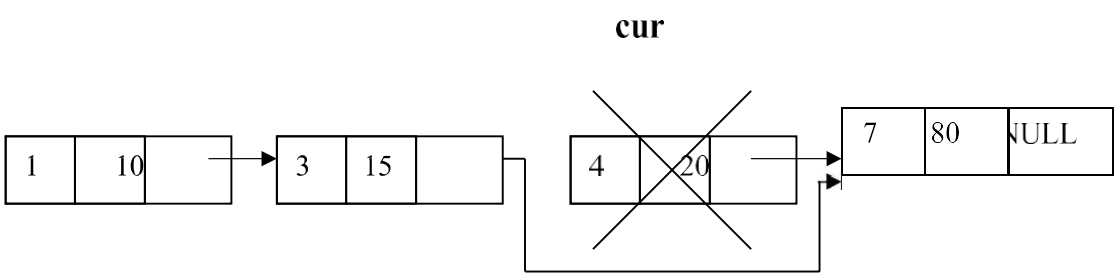



```
void dictionary::insert_d( )
{
node *p,*curr,*prev;
            cout<<"Enter an key and value to be inserted:";
            cin>>k;
           cin>>data;
```

```
        p=new node;
        p->key=k;
       p->value=data;
       p->next=NULL;
        if(head==NULL)
                head=p;
       else
       {
              curr=head;
               while((curr->key<p->key)&&(curr->next!=NULL))
              {
                      prev=curr;
                      curr=curr->next;
                }
               if(curr->next==NULL)
              {
                      if(curr->key<p->key)
                        {
                              curr->next=p;
                               prev=curr;
                      }
                      else
                      {
                             p->next=prev->next;
                              prev->next=p;
                      }
               }
               else
              {
                      p->next=prev->next;
                      prev->next=p;
              }
              cout<<"\nInserted into dictionary Sucesfully.... \n";
       }
}
```

The delete operation:

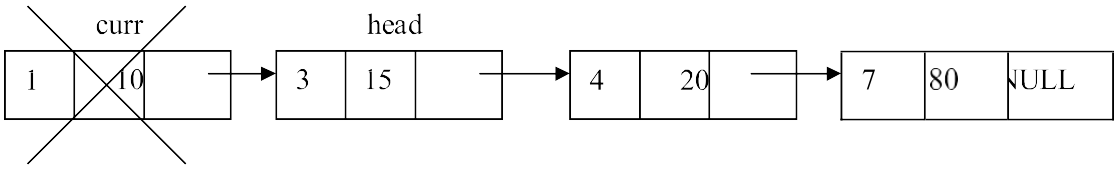
Case 1: Initially assign 'head' node as 'curr' node.Then ask for a key value of the node which is to be deleted. Then starting from head node key value of each jode is cked and compared with the desired node's key value. We will get node which is to be deleted in variable 'curr'. The node given by variable 'prev' keeps track of previous node of 'cuu' node. For eg, delete node with key value 4 then



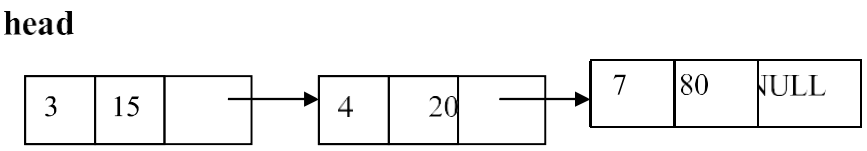
Case 2:

If the node to be deleted is head node
i.e.. if(curr==head)

Then, simply make 'head' node as next node and delete 'curr'



Hence the list becomes




```
void dictionary::delete_d( )
{
node*curr,*prev;
        cout<<"Enter key value that you want to delete...";
        cin>>k;
        if(head==NULL)
            cout<<"\ndictionary is Underflow";
        else
          {   curr=head;
              while(curr!=NULL)
              {
              if(curr->key==k)
                      break;
              prev=curr;
              curr=curr->next;
              }
          }
        if(curr==NULL)
            cout<<"Node not found...";
        else
          {
             if(curr==head)
```

```
                    head=curr->next;
                else
                    prev->next=curr->next;
            delete curr;
            cout<<"Item deleted from dictionary...";
            }
 }
```

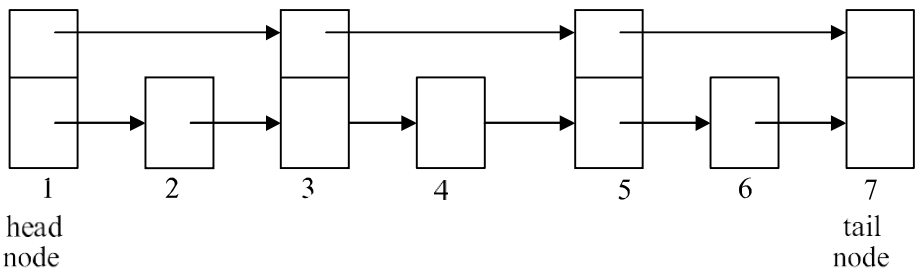
The length operation:

```
int dictionary::length()
{
struct node *curr;
int count;
        count=0;
        curr=head;
        if(curr==NULL)
        {
                cout<<"The list is empty";
                return 0;
        }
        while(curr!=NULL)
        {
                count++;
                cur=curr->next;
        }
return count;
 }
```

SKIP LIST REPRESENTATION

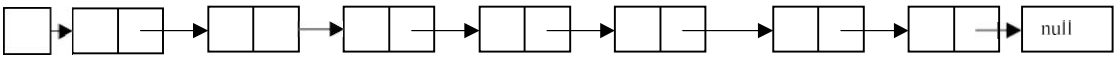
Skip list is a variant list for the linked list. Skip lists are made up of a series of nodes connected one after the other. Each node contains a key and value pair as well as one or more references, or pointers, to nodes further along in the list. The number of references each node contains is determined randomly. This gives skip lists their probabilistic nature, and the number of references a node contains is called its node level.

There are two special nodes in the skip list one is head node which is the starting node of the list and tail node is the last node of the list

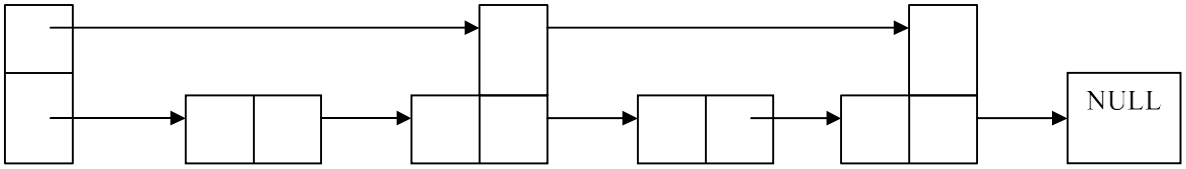


The skip list is an efficient implementation of dictionary using sorted chain. This is because in skip list each node consists of forward references of more than one node at a time.

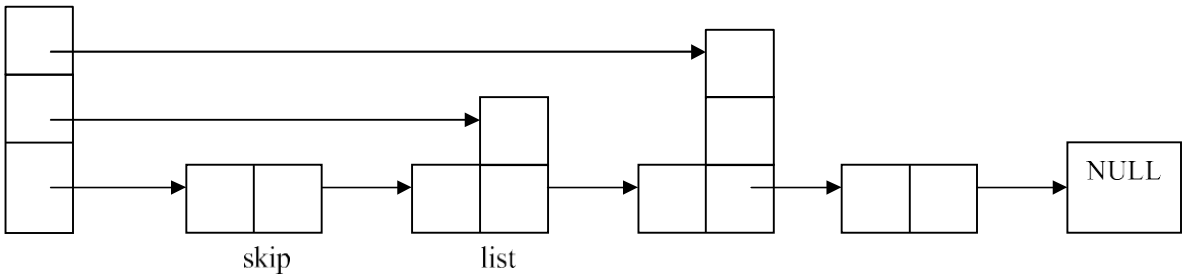
Eg:



Now to search any node from above given sorted chain we have to search the sorted chain from head node by visiting each node. But this searching time can be reduced if we add one level in every alternate node. This extra level contains the forward pointer of some node. That means in sorted chain come nodes can holds pointers to more than one node.



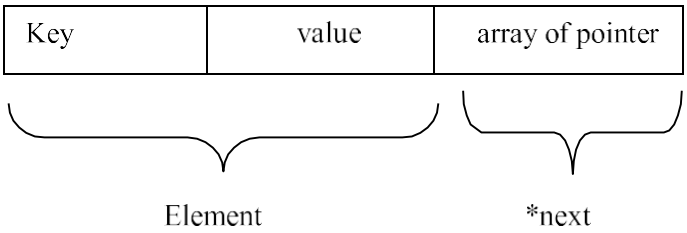
If we want to search node 40 from above chain there we will require comparatively less time. This search again can be made efficient if we add few more pointers forward references.



Node structure of skip list:

```
template <class K, class E>
struct skipnode
{
        typedef pair<const K,E> pair_type;
        pair_type element;
        skipnode<K,E> **next;
        skipnode(const pair_type &New_pair, int MAX):element(New_pair)
        {
                next=new skipnode<K,E>*[MAX];
        }
};
```

The individual node looks like this:



Searching:

The desired node is searched with the help of a key value.

```
template<class K, class E>
skipnode<K,E>* skipLst<K,E>::search(K& Key_val)
{
skipnode<K,E>* Forward_Node = header;
for(int i=level;i>=0;i--)
{
while (Forward_Node->next[i]->element.key < key_val)
Forward_Node = Forward_Node->next[i];
last[i] = Forward_Node;
}
return Forward_Node->next[0];
}
```

Searching for a key within a skip list begins with starting at header at the overall list level and moving forward in the list comparing node keys to the key_val. If the node key is less than the key_val, the search continues moving forward at the same level. If o the other hand, the node key is equal to or greater than the key_val, the search drops one level and continues forward. This process continues until the desired key_val has been found if it is present in the skip list. If it is not, the search will either continue at the end of the list or until the first key with a value greater than the search key is found.

Insertion:

There are two tasks that should be done before insertion operation:

1. Before insertion of any node the place for this new node in the skip list is searched. Hence before any insertion to take place the search routine executes. The last[] array in the search routine is used to keep track of the references to the nodes where the search, drops down one level.
2. The level for the new node is retrieved by the routine randomelevel()

```
template<class K,class E>
void skipLst<K,E>::insert(pair<K,E>& New_pair)
{
if(New_pair.key >= tailkey)
{
cout<<"Key is too large";
}

skipNode<K,E>* temp = search(New_pair.key);
if(temp->element.key == New_pair.key)
```

```
{
temp->element.value=New_pair.value;
return;
}

if*New_Level > levels)
{
New_Level = ++levels;
last[New_Level] = header;
}

skipNode<K,E> *newNode = new skipNode<K,E>(New_pair, New_Level+1);

for(int i=0;i<=New_Level;i++)
{
newNode->next[i] = last[i]->next[i];
last[i]->next[i] = newNode;
}
len++;
return;
}
```

Determining the level of each node:

```
template <class K, class E>
int skipLst<K,E>::randomlevel()
{
int lvl=0;
while(rand() <= Lvl_No)
lvl=lvl+1;
if(lvl<=MaxLvl)
return lvl;
else
return MaxLvl;
}
```

Deletion:

First of all, the deletion makes use of search algorithm and searches the node that is to be deleted. If the key to be deleted is found, the node containing the key is removed.

```
template<class K, class E>
void skipLst<K,E>::delet(K& Key_val)
{
if(key_val>=tailKey)
return;
skipNode<K,E>* temp = search(Key_val);
if(temp->elemnt.key != Key_val)
return;

for(int i=0;i<=levels;i++)
```

```
{
if(last[i]->next[i] == temp)
last[i]=>next[i] = temp->next[i];
}

while(level>0 && header->next[level] == tail)
levels--;
delete temp;
len--;
}
```

HASH TABLE REPRESENTATION

- 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 value. Hence every entry in the hash table is associated with some key.
- Using the hash key the required piece of data can be searched in the hash table by few or more key comparisons. The searching time is then dependent upon the size of the hash table.
- The effective representation of dictionary can be done using hash table. We can place the dictionary entries in the hash table using hash function.

HASH FUNCTION

- Hash function is a function which is used to put the data in the hash table. Hence one can use the same hash function to retrieve the data from the hash table. Thus hash function is used to implement the hash table.
- The integer returned by the hash function is called hash key.

For example: Consider that we want place some employee records in the hash table The record of employee is placed with the help of key: employee ID. The employee ID is a 7 digit number for placing the record in the hash table. To place the record 7 digit number is converted into 3 digits by taking only last three digits of the key.

If the key is 496700 it can be stored at 0^{th} position. The second key 8421002, the record of those key is placed at 2^{nd} position in the array.
Hence the hash function will be- H(key) = key%1000
Where key%1000 is a hash function and key obtained by hash function is called hash key.

- **Bucket and Home bucket:** The hash function H(key) is used to map several dictionary entries in the hash table. Each position of the hash table is called bucket.

The function H(key) is home bucket for the dictionary with pair whose value is key.

TYPES OF HASH FUNCTION

There are various types of hash functions that are used to place the record in the hash table-

1. Division Method: The hash function depends upon the remainder of division.

Typically the divisor is table length.
For eg; If the record 54, 72, 89, 37 is placed in the hash table and if the table size is 10 then

h(key) = record % table size

54%10=4
72%10=2
89%10=9
37%10=7

0	
1	
2	72
3	
4	54
5	
6	
7	37
8	
9	89

2. Mid Square:

In the mid square method, the key is squared and the middle or mid part of the result is used as the index. If the key is a string, it has to be preprocessed to produce a number.
Consider that if we want to place a record 3111 then

$3111^2 = 9678321$
for the hash table of size 1000
H(3111) = 783 (the middle 3 digits)

3. Multiplicative hash function:

The given record is multiplied by some constant value. The formula for computing the hash key is-

H(key) = floor(p *(fractional part of key*A)) where p is integer constant and A is constant real number.

Donald Knuth suggested to use constant A = 0.61803398987

If key 107 and p=50 then

H(key) = floor(50*(107*0.61803398987))
= floor(3306.4818458045)
= 3306
At 3306 location in the hash table the record 107 will be placed.

4. Digit Folding:

The key is divided into separate parts and using some simple operation these parts are combined to produce the hash key.
For eg; consider a record 12365412 then it is divided into separate parts as 123 654 12 and these are added together

H(key) = 123+654+12
= 789
The record will be placed at location 789

5. Digit Analysis:

The digit analysis is used in a situation when all the identifiers are known in advance. We first transform the identifiers into numbers using some radix, r. Then examine the digits of each identifier. Some digits having most skewed distributions are deleted. This deleting of digits is continued until the number of remaining digits is small enough to give an address in the range of the hash table. Then these digits are used to calculate the hash address.

COLLISION

the hash function is a function that returns the key value using which the record can be placed in the hash table. Thus this function helps us in placing the record in the hash table at appropriate position and due to this we can retrieve the record directly from that location. This function need to be designed very carefully and it should not return the same hash key address for two different records. This is an undesirable situation in hashing.

Definition: The situation in which the hash function returns the same hash key (home bucket) for more than one record is called collision and two same hash keys returned for different records is called synonym.

Similarly when there is no room for a new pair in the hash table then such a situation is called overflow. Sometimes when we handle collision it may lead to overflow conditions. Collision and overflow show the poor hash functions.

For example,

Consider a hash function.

H(key) = recordkey%10 having the hash table size of 10

The record keys to be placed are

131, 44, 43, 78, 19, 36, 57 and 77

131%10=1
44%10=4
43%10=3
78%10=8
19%10=9
36%10=6
57%10=7
77%10=7

0	
1	131
2	
3	43
4	44
5	
6	36
7	57
8	78
9	19

Now if we try to place 77 in the hash table then we get the hash key to be 7 and at index 7 already the record key 57 is placed. This situation is called **collision**. From the index 7 if we look for next vacant position at subsequent indices 8.9 then we find that there is no room to place 77 in the hash table. This situation is called **overflow**.

COLLISION RESOLUTION TECHNIQUES

If collision occurs then it should be handled by applying some techniques. Such a technique is called collision handling technique.

1. Chaining
2. Open addressing (linear probing)
3. Quadratic probing
4. Double hashing
5. Double hashing
6. Rehashing

CHAINING

In collision handling method chaining is a concept which introduces an additional field with data i.e. chain. A separate chain table is maintained for colliding data. When collision occurs then a linked list(chain) is maintained at the home bucket.

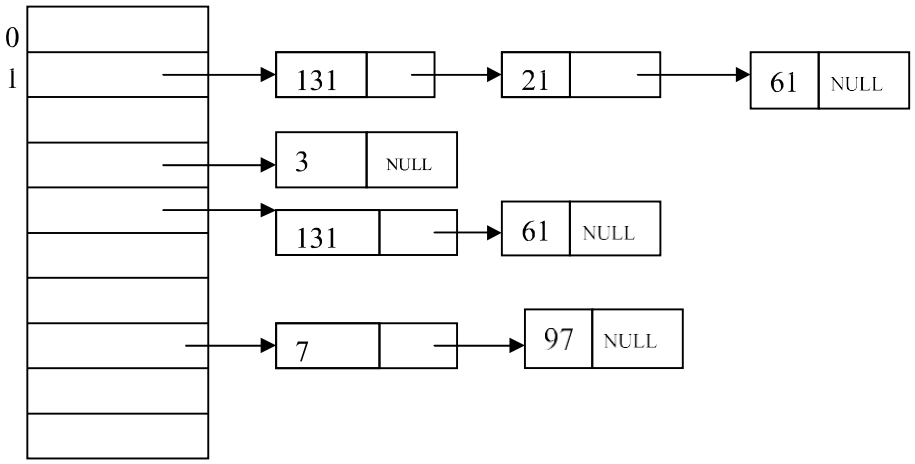
For eg;

Consider the keys to be placed in their home buckets are
131, 3, 4, 21, 61, 7, 97, 8, 9

then we will apply a hash function as H(key) = key % D

Where D is the size of table. The hash table will be-

Here D = 10



A chain is maintained for colliding elements. for instance 131 has a home bucket (key) 1. similarly key 21 and 61 demand for home bucket 1. Hence a chain is maintained at index 1.

OPEN ADDRESSING – LINEAR PROBING

This is the easiest method of handling collision. When collision occurs i.e. when two records demand for the same home bucket in the hash table then collision can be solved by placing the second record linearly down whenever the empty bucket is found. When use linear probing (open addressing), the hash table is represented as a one-dimensional array with indices that range from 0 to the desired table size-1. Before inserting any elements into this table, we must initialize the table to represent the situation where all slots are empty. This allows us to detect overflows and collisions when we inset elements into the table. Then using some suitable hash function the element can be inserted into the hash table.

For example:

Consider that following keys are to be inserted in the hash table

131, 4, 8, 7, 21, 5, 31, 61, 9, 29

Initially, we will put the following keys in the hash table.
We will use Division hash function. That means the keys are placed using the formula

H(key) = key % tablesize
H(key) = key % 10

For instance the element 131 can be placed at

H(key) = 131 % 10
= 1

Index 1 will be the home bucket for 131. Continuing in this fashion we will place 4, 8, 7.

Now the next key to be inserted is 21. According to the hash function

H(key)=21%10
H(key) = 1

But the index 1 location is already occupied by 131 i.e. collision occurs. To resolve this collision we will linearly move down and at the next empty location we will prob the element. Therefore 21 will be placed at the index 2. If the next element is 5 then we get the home bucket for 5 as index 5 and this bucket is empty so we will put the element 5 at index 5.

Index	Key
0	NULL
1	131
2	NULL
3	NULL
4	4
5	NULL
6	NULL
7	7
8	8
9	NULL

Key
NULL
131
21
NULL
4
5
NULL
7
8
NULL

Key
NULL
131
21
31
4
5
61
7
8
NULL

after placing keys 31, 61

The next record key is 9. According to decision hash function it demands for the home bucket 9. Hence we will place 9 at index 9. Now the next final record key 29 and it hashes a key 9. But home bucket 9 is already occupied. And there is no next empty bucket as the table size is limited to index 9. The overflow occurs. To handle it we move back to bucket 0 and is the location over there is empty 29 will be placed at 0^{th} index.

Problem with linear probing:

One major problem with linear probing is primary clustering. Primary clustering is a process in which a block of data is formed in the hash table when collision is resolved.

19%10 = 9
18%10 = 8
39%10 = 9
29%10 = 9
8%10 = 8

cluster is formed

rest of the table is empty

this cluster problem can be solved by quadratic probing.

Key
39
29
8
18
19

QUADRATIC PROBING:

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This method uses following formula.

$$H(key) = (Hash(key) + i^2) \% m)$$

where m can be table size or any prime number.

for eg; If we have to insert following elements in the hash table with table size 10:

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

37 % 10 = 7
90 % 10 = 0
55 % 10 = 5
22 % 10 = 2
11 % 10 = 1

Now if we want to place 17 a collision will occur as 17%10 = 7 and bucket 7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.

0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	
9	

$H_i(key) = (Hash(key) + i^2) \% m$

Consider i = 0 then
$(17 + 0^2) \% 10 = 7$

$(17 + 1^2) \% 10 = 8$, when i =1

The bucket 8 is empty hence we will place the element at index 8.
Then comes 49 which will be placed at index 9.

49 % 10 = 9

0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	49
9	

Now to place 87 we will use quadratic probing.

$(87 + 0) \% 10 = 7$
$(87 + 1) \% 10 = 8$… but already occupied
$(87 + 2^2) \% 10 = 1$.. already occupied
$(87 + 3^2) \% 10 = 6$

It is observed that if we want place all the necessary elements in the hash table the size of divisor (m) should be twice as large as total number of elements.

0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	49
9	

DOUBLE HASHING

Double hashing is 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.

Consider the following elements to be placed in the hash table of size 10
37, 90, 45, 22, 17, 49, 55
Initially insert the elements using the formula for $H_1(key)$.
Insert 37, 90, 45, 22

$H_1(37) = 37 \% 10 = 7$
$H_1(90) = 90 \% 10 = 0$
$H_1(45) = 45 \% 10 = 5$
$H_1(22) = 22 \% 10 = 2$
$H_1(49) = 49 \% 10 = 9$

Key
90
22
45
37
49

Now if 17 to be inserted then

$H_1(17) = 17 \% 10 = 7$
$H_2(key) = M - (key \% M)$

Here M is prime number smaller than the size of the table. Prime number smaller than table size 10 is 7

Hence M = 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 ha v
jumps. Therefore the 17 will be placed at index 1.

Key
90
17
22
45
37
49

Now to insert number 55

$H_1(55) = 55 \% 10 = 5 \longrightarrow$ Collision

$H_2(55) = 7-(55 \% 7)$
$= 7 - 6 = 1$

That means we have to take one jump from index 5 to place 55.
Finally the hash table will be -

Key
90
17
22
45
55
37
49

Comparison of Quadratic Probing & Double Hashing

The double hashing requires another hash function whose probing efficiency is same as some another hash function required when handling random collision.

The double hashing is more complex to implement than quadratic probing. The quadratic probing is fast technique than double hashing.

REHASHING

Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating a new table. It is preferable is the total size of table is a prime number. There are situations in which the rehashing is required.

- When table is completely full
- With quadratic probing when the table is filled half.
- When insertions fail due to overflow.

In such situations, we have to transfer entries from old table to the new table by re computing their positions using hash functions.

Consider we have to insert the elements 37, 90, 55, 22, 17, 49, and 87. the table size is 10 and will use hash function.,

H(key) = key mod tablesize

37 % 10 = 7
90 % 10= 0
55 % 10 = 5
22 % 10 = 2
17 % 10 = 7 Collision solved by linear probing
49 % 10 = 9

Now this table is almost full and if we try to insert more elements collisions will occur and eventually further insertions will fail. Hence we will rehash by doubling the table size. The old table size is 10 then we should double this size for new table, that becomes 20. But 20 is not a prime number, we will prefer to make the table size as 23. And new hash function will be

H(key) key mod 23

37 % 23 = 14
90 % 23 = 21
55 % 23 = 9
22 % 23 = 22
17 % 23 = 17
49 % 23 = 3
87 % 23 = 18

0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	49
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	

Now the hash table is sufficiently large to accommodate new insertions.

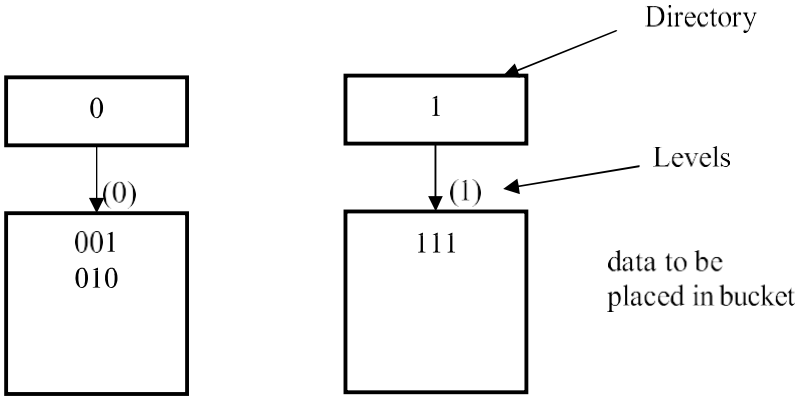
Advantages:

1. This technique provides the programmer a flexibility to enlarge the table size if required.
2. Only the space gets doubled with simple hash function which avoids occurrence of collisions.

EXTENSIBLE HASHING

- Extensible hashing 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 parenthesis.

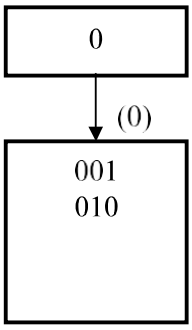
For eg:



- 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.

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

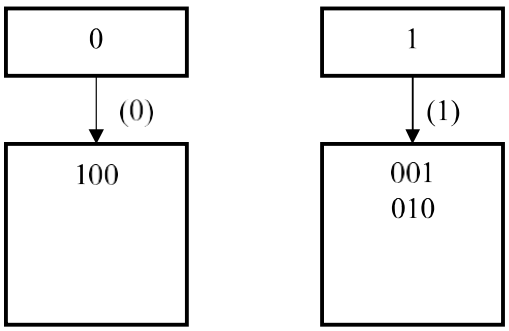


1 = 001

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

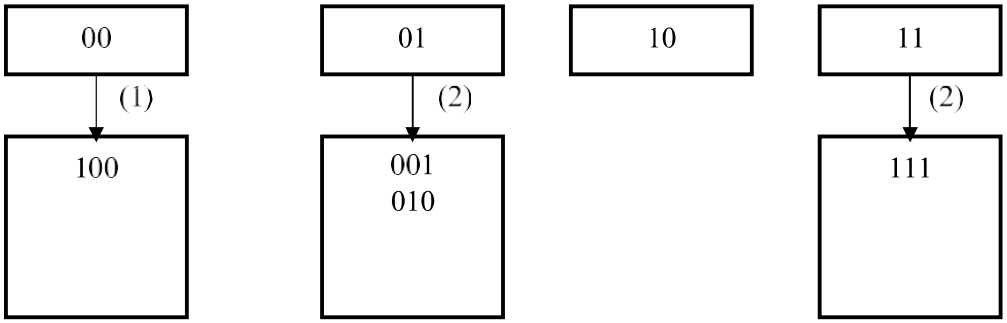
5 = 101

Based on last bit the data is inserted.

Step 2: Insert 7

7 = 111

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