UNIT - II

Dictionaries

- Dictionary is a collection of Pairs of Key and value where every Value is associated with the corresponding Key.

- As well as the data (values), that the dictionary holds, there is also a key that is used for searching and finding the required values. the elements of the dictionary are represented by Pairs(key, value), where the key is used for searching.

Ex:-

- The Library book database is Prepared which is actually a dictionary when a book is issued to the student that book entry is deleted from the database and on returning of the book to the Library the corresponding book entry is added in the book database. if a strudent demands for some specific book then its entry is searched in the book database.
- Water Park, there is usually a place where you can ~~where~~ leave your outdoor clothing.
- Wallet car parking

- when using the Dictionary, the key may not just be a number, but any other type of object.

- In this case when we have a key (number), we could implement this type of structure as a regular array. In this scenario the set of keys is already known - these are the numbers from 0 to n, where n represents the size of the array.

- when using dictionaries, the set of keys usually is a randomly chosen set of values like real numbers (or) strings. the only restriction is that we can distinguish one key from the other. Later we will take a look at some additional requirements for the keys that are needed for the different kinds of implementations.

- For every Key in the dictionary, there is a corresponding value. one key can hold only one value. The aggregation of all the Pairs (Key, Value) represents the dictionary.

- Dictionary is a data structure using these bellow mentioned Operations can be Performed efficiently:

    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.

Applications:
- Student registration application
- Telephone directory
- word dictionary
- symbol table used in compiler

Linear List Representation :-

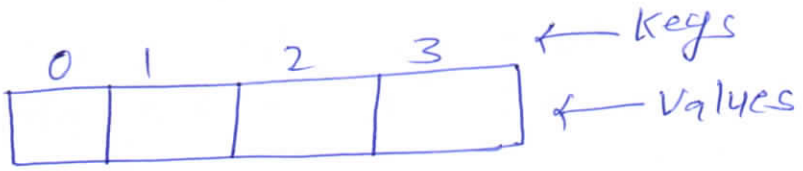
- The dictionary can be represented as a linear List. the linear List is a collection of ~~Pair~~ Key and Value

There are 2 methods of representing linear List:

1. Sorted Array: An array datastructure is used to implement the Dictionary.
2. Sorted chain: A linked List data structure is used to implement the dictionary.

Sorted Array:

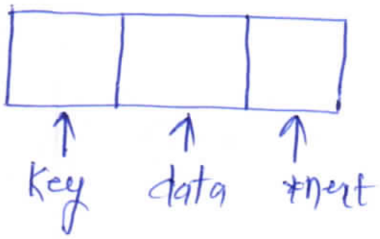
int Arr[4]



- if Keys are integer numbers and follows sequential order then use sorted Array for the dictionary implementation for that data. otherwise use the sorted chain (Lists).

Sorted chain:-

node structure



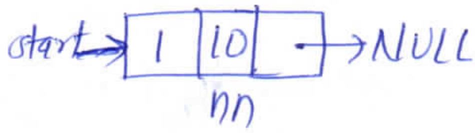
```
struct node
{
 int data,key;
 struct node *next;
} *start=NULL;
```

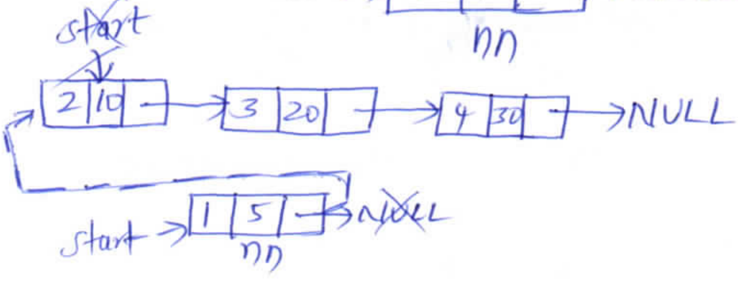
Insertion:

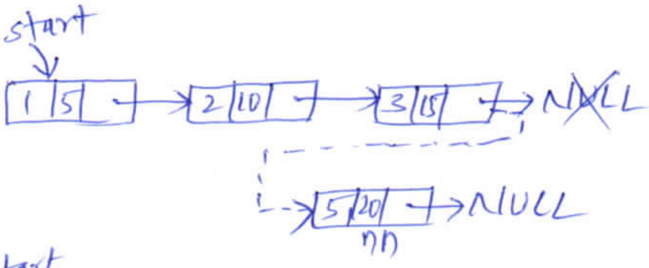
```
struct node *nn;
nn=(struct node *)malloc(sizeof
                        (struct node));
printf("enter key value");
scanf("%d",&nn->key); 1
printf("enter data for node");
scanf("%d",&nn->data); 10
nn->next=NULL;
```

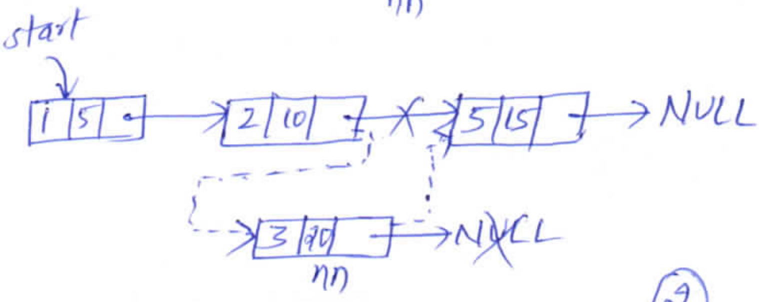
```
if(start==NULL) //List is empty
    start=nn;

else
 {struct node t=start, t1=start->next;
   if(t->key > nn->key)
     {
       nn->next = t;
       start = nn;
     }
   else
     {
       while(t1->key ≯ nn->key && t1->next!=NULL)
          { t=t->next;
            t1 = t1->next;
          }
       if(t1->next==NULL)
           t1->next=nn;
       else
          { nn->next=t1;
            t->next=nn;
 }        }
```



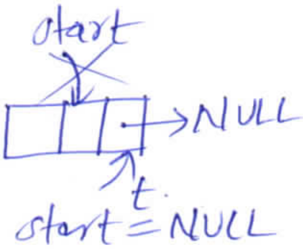


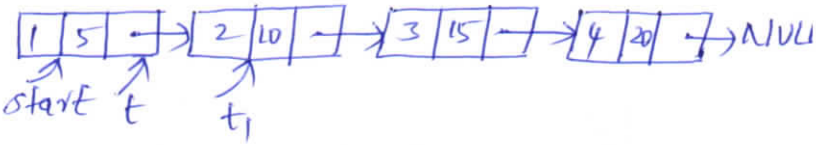




Deletion:-

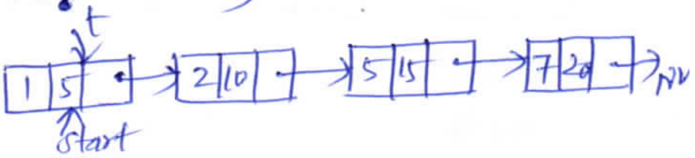
```
if (start == NULL) //empty
  Printf(" Dictionary is empty");
else
 { int a;
   Printf("enter Key, for which node you want to delete");
   scanf("%d", &a);
   struct node *t = start, *t1 = start→next;
    if(t→next == NULL) //one node
    { if( t→key == a)
       { start = NULL;
       } free(t);
      else
        Printf("node is not available in Dictionary");
    }
   Else if (start→key==a) // first element to delete
    {   {start=start→next;
          t→next=NULL; free(t);}
else
  while~~if~~(t1 → key == a && t1→next != NULL) // search which element to delete
      { t = t→next;
        t1 = t1→next;
      }
     ~~else~~
      {
       if(t1→next==NULL)
        Printf("node is not available in Dictionary");
       else
         {  t→next = t1→next;
            t1→next=NULL;
         } free (t);
      }
    }
 }
```





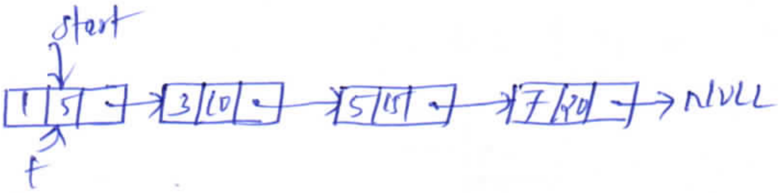
Searching

```
if(start==NULL)
   Printf("Dictionary is Empty");
else
  { int a;
    struct node *t=start;
    Printf("enter the Key to find the data");
    scanf("%d", &a);
     while(t->key != a && t->next != NULL)
         t= t->next;
       if(t->next == NULL)
        Printf("Key is not in the Dictionary");
       else
         Printf("Key is exists in the Dictionary and data is:", t->data);
  }
```



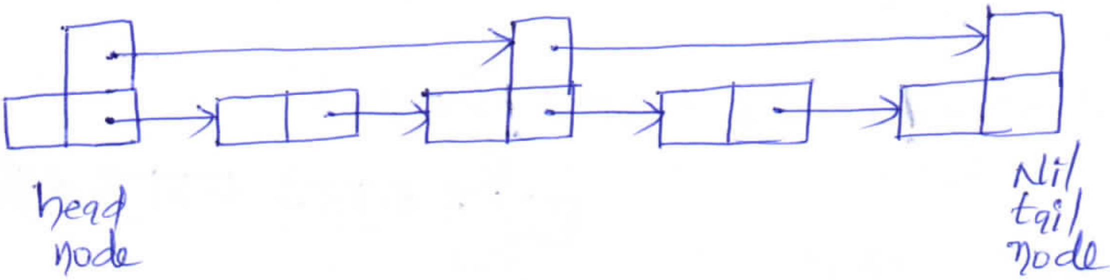
Display

```
 if(start==NULL)
   Printf("Dictionary is Empty");
  else
  { struct node *t=start
     while(t->next!=NULL)
        { Printf("Dictionary Key and Data is %d %d", t->key, t->data);
            t=t->next;
        }
  }
```



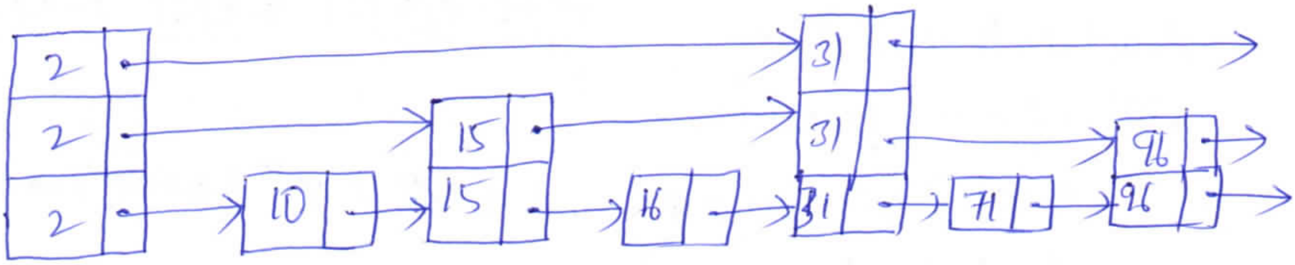
skip List

- skip list is a variant list for the linked List. skip lists ale 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 ale 2 special nodes in the skiplist 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 sorked chain This is because in skip list each node Consists of forward references of more than one node at a time

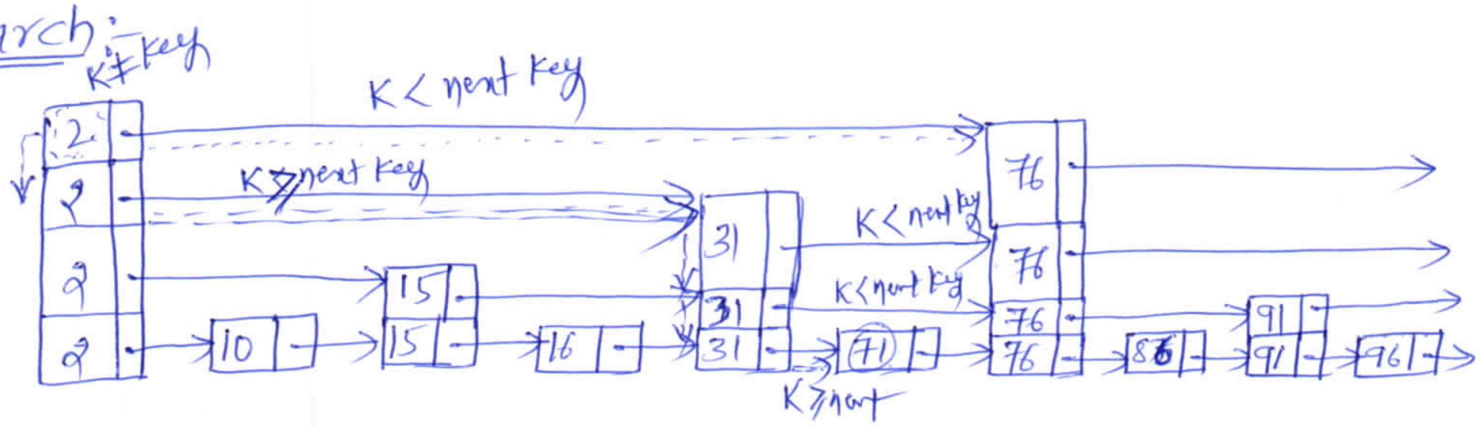
Ex: Perfect skip List



- Keys in sorted order, and O(logn) levels.
- each higher level contains 1/2 the elements of the level below it
- Nodes ale of variable size which contains between 1 and O(logn) Pointers.

Operations:

Search:



⇒ Find 71 K

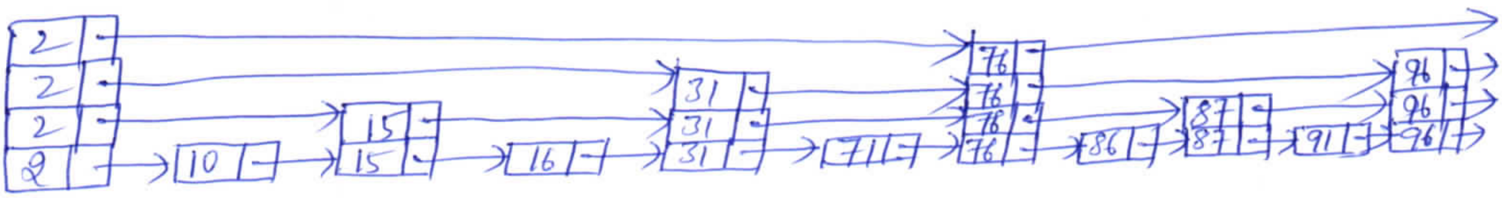
⇒ Find 96

when search for K:

- if K = key, done
- if K < next key, go down a level
- if K ≥ next key, go right

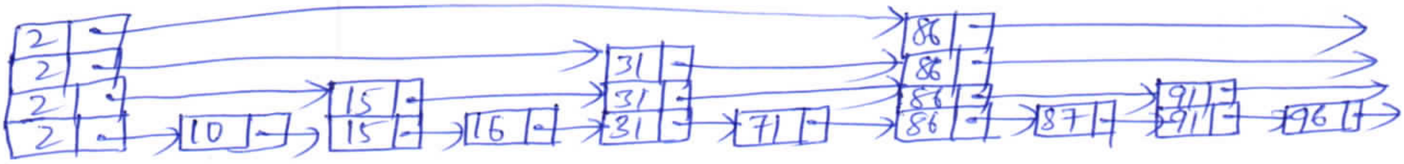
Insertion:

- Insert 87



Deletion:

- Delete 76



```
/* Skip Lists: A Probabilistic Alternative to Balanced Trees */

#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#define SKIPLIST_MAX_LEVEL 6

typedef struct snode {

    int key;
    int value;
    struct snode **forward;
} snode;

 typedef struct skiplist {

    int level;
    int size;
    struct snode *header;
} skiplist;

  skiplist *skiplist_init(skiplist *list) {

    int i;
    snode *header = (snode *) malloc(sizeof(struct snode));
    list->header = header;
    header->key = INT_MAX;
    header->forward = (snode **) malloc(
            sizeof(snode*) * (SKIPLIST_MAX_LEVEL + 1));
    for (i = 0; i <= SKIPLIST_MAX_LEVEL; i++) {
        header->forward[i] = list->header;
    }

    list->level = 1;
    list->size = 0;
    return list;
}

  static int rand_level() {

    int level = 1;
    while (rand() < RAND_MAX / 2 && level < SKIPLIST_MAX_LEVEL)
        level++;
    return level;
}

int skiplist_insert(skiplist *list, int key, int value) {

    snode *update[SKIPLIST_MAX_LEVEL + 1];
    snode *x = list->header;
    int i, level;
    for (i = list->level; i >= 1; i--) {
        while (x->forward[i]->key < key)
            x = x->forward[i];
        update[i] = x;
    }
    x = x->forward[1];
    if (key == x->key) {

        x->value = value;
        return 0;
    } else {

        level = rand_level();
        if (level > list->level) {

            for (i = list->level + 1; i <= level; i++) {
                update[i] = list->header;
            }
            list->level = level;
        }

        x = (snode *) malloc(sizeof(snode));
        x->key = key;
        x->value = value;
        x->forward = (snode **) malloc(sizeof(snode*) * (level + 1));
        for (i = 1; i <= level; i++) {
            x->forward[i] = update[i]->forward[i];
            update[i]->forward[i] = x;
        }
    }
    return 0;
}

snode *skiplist_search(skiplist *list, int key) {

    snode *x = list->header;
    int i;
    for (i = list->level; i >= 1; i--) {

        while (x->forward[i]->key < key)
            x = x->forward[i];
    }

    if (x->forward[1]->key == key) {

        return x->forward[1];
    } else {

        return NULL;
    }
    return NULL;
 }

 static void skiplist_node_free(snode *x) {

    if (x) {
        free(x->forward);
        free(x);
    }
 }

int skiplist_delete(skiplist *list, int key) {

    int i;
    snode *update[SKIPLIST_MAX_LEVEL + 1];
    snode *x = list->header;
    for (i = list->level; i >= 1; i--) {

        while (x->forward[i]->key < key)
            x = x->forward[i];
        update[i] = x;
    }

   x = x->forward[1];
    if (x->key == key) {

        for (i = 1; i <= list->level; i++) {
            if (update[i]->forward[i] != x)
                break;
            update[i]->forward[1] = x->forward[i];
        }

        skiplist_node_free(x);
        while (list->level > 1 && list->header->forward[list->level] == list->header)
            list->level--;
        return 0;
    }
    return 1;
 }
 static void skiplist_dump(skiplist *list) {
    snode *x = list->header;
    while (x && x->forward[1] != list->header) {
        printf("%d[%d]->", x->forward[1]->key, x->forward[1]->value);
        x = x->forward[1];
    }
    printf("NIL\n");
 }
 int main() {
    int arr[] = { 3, 6, 9, 2, 11, 1, 4 }, i;
    skiplist list;
    skiplist_init(&list);
    printf("Insert:--------------------\n");

    for (i = 0; i < sizeof(arr) / sizeof(arr[0]); i++) {
        skiplist_insert(&list, arr[i], arr[i]);
    }
    skiplist_dump(&list);
    printf("Search:--------------------\n");
    int keys[] = { 3, 4, 7, 10, 111 };

    for (i = 0; i < sizeof(keys) / sizeof(keys[0]); i++) {

        snode *x = skiplist_search(&list, keys[i]);
        if (x) {
            printf("key = %d, value = %d\n", keys[i], x->value);
        } else {
            printf("key = %d, not fuound\n", keys[i]);
        }
    }
    printf("Search:--------------------\n");
    skiplist_delete(&list, 3);
    skiplist_delete(&list, 9);
    skiplist_dump(&list);
    return 0;
 }
```

Hashing

- In all search techniques like linear search, binary search and search trees, the time required to search an element is depends on the total number of element in that data structure. in all these search techniquies, as the number of element are increased the time required to search an element also increased linearly.

- Hashing is another approach in which time required to search an element doesn't depend on the number of elements.

- using hashing data structure, an element is searched with constant time complexity.

- Hashing is an effective way to reduce the number of comparisions to search an element in a data structure.

—* Hashing is the Process of indexing and retriving element in a data structure to Provide faster way of finding the element using the hash key

- hash key is a value which Provides the index value where the actual data is likely to store in the data structure.

Hash Table :-

- In this data structure, we use a concept called Hash Table to store data.

- all the data values are inserted into the hash table based on the hash key value. hash key value is used to map the data with index in the hash table.

- And the hash key is generated for every data using a hash function. that means every entry in the hash table is based on the key value generated using a hash function.

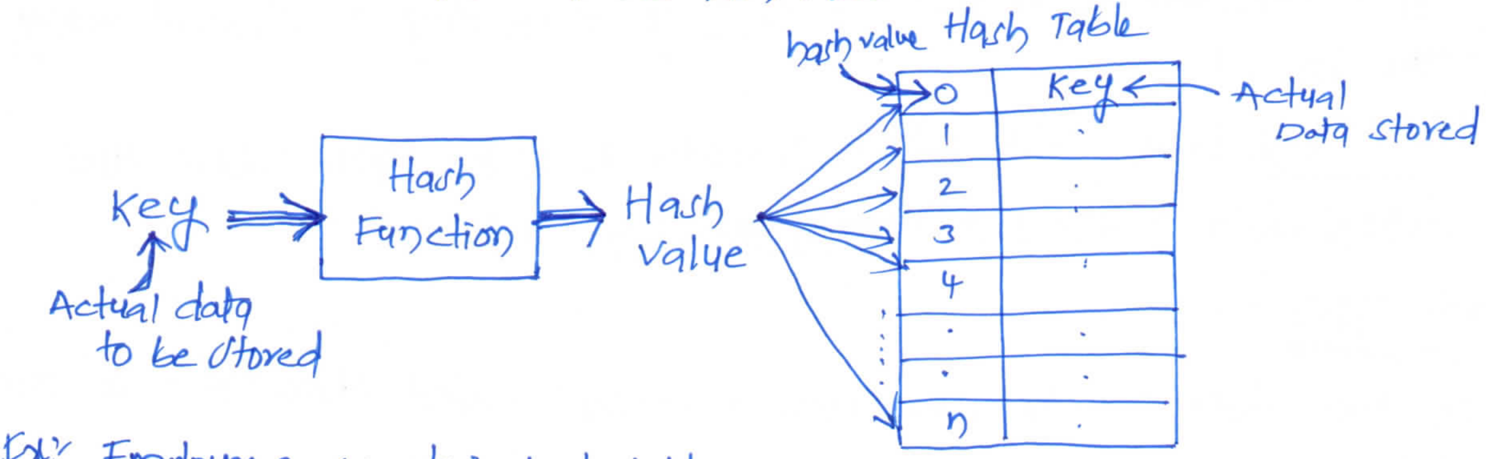
* hash tables is just an array which maps a Key into the data structure with the help of hash function such that insertion, deletion and search operations can be Performed with constant time complexity (i.e. O(1)).

→ ~~The searching~~

Hash Function:

- hash tables are used to Perform the operations like insertion, deletion and search very quickly in data structure with constant time.
- Generally, every hash table make use of a function, which will call the hash function to map the data into the hash table.

- hash function is a function which takes a Piece of data (i.e. key) as input and outputs an integer (i.e. hash value) which maps the data to a Particular index in the hash table



Ex: Employees records in hash table

- Employee ID contains 7 digits
- but key taken only last 3 digits

Hash function:
H(Key) = Key % 1000

Hash Table

	EMP_ID	Record
0	6243000	
1	5786001	
2	4796002	
3	7985003	
4	8975004	
.	.	
.	.	
998	2347998	
999	5789999	

Types of Hash Functions

1) Division Method:

The hash function depends upon the remainder of division. The divisor is table length

Ex: if the record 54, 72, 89, 37 is to be placed in the hash table and the table size is 10.

$H(key) = (record) \% (table\ size)$

$= 54 \% 10 = 4$

$= 72 \% 10 = 2$

$= 89 \% 10 = 9$

$= 37 \% 10 = 7$

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

2) Mid square:

- 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

Ex: 1-record = 15

$H(key) = (15)^2 = 2\textcircled{2}5 = 2$

2-record = 17

$= (17)^2 = 2\textcircled{89} = 8$

0	
1	
2	15
3	
4	
5	
6	
7	
8	17
9	

3) Multiplicative hash function:

- The given record is multiplied by some constant value

hash function:

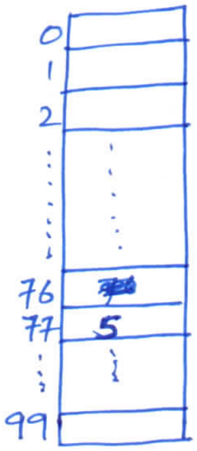
$$H(Key) = Floor(P * (fractional\ Part\ of\ Key * A))$$

where P is integer constant
A is constant real number

- Donald Knuth suggested to use constant $A = 0.61803398987$

if key 5 and $P = 25$ then

$$H(Key) = Floor(25 * (5 * 0.61803398987))$$
$$= Floor(77.25424873375)$$
$$= 77$$



4). Digit Folding:-

- The key is divided into separate Parts and using some simple opelation these Parts ale combined to Produle the hash key.

Ex:- A record 12365412 then it divided into separate Parts ale 123 654 12 and these ale added together

$$H(Key) = 123 + 654 + 12 = 789$$

- The record will be placed at location 789 in the hash table index

Collision

- The hash function returns the same hash key for more than one record is called collision and 2 same hash keys returned for different records is called synonym.

- When there is no room for a new pair in the hash table then such a situation is called overflow.

- collision and overflow show the Poor hash functions

Ex:- H(key) = record % 10

31, 44, 43, 78, 19, 36, 57

⇒ 77

→ Now 77 hash key is index 7 is already the record key 57 is place.
- this situation is called collision

0	
1	31
2	
3	43
4	44
5	
6	36
7	57
8	78
9	19

→ From the index 7 if we look for next vacant Passion at subsequent indices 8, 9 then we find that there is no room to place 77 in the hash table this is called overflow.

Overflow Handling :-

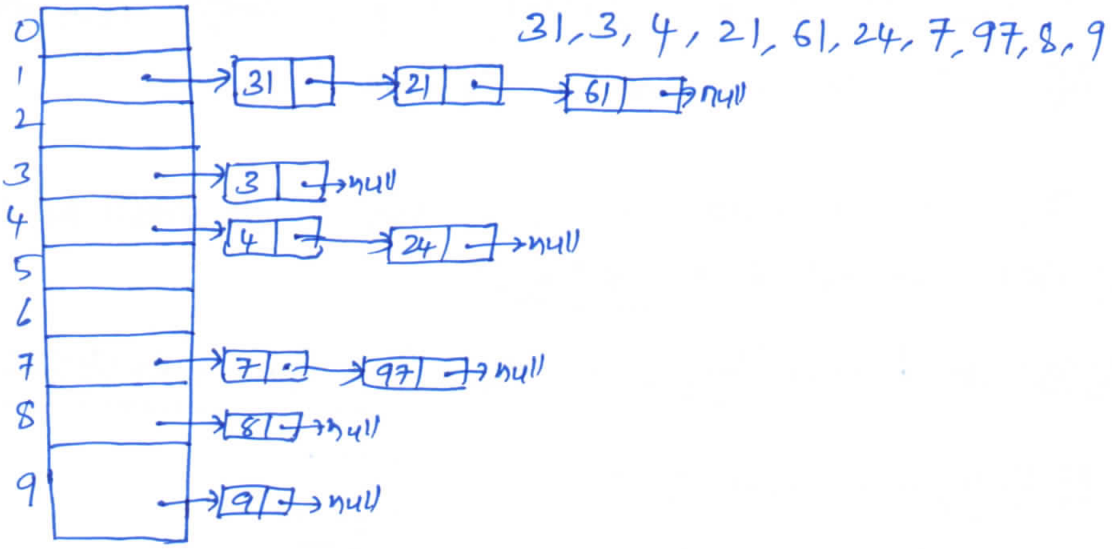
- We can handle overflow in Hashing to following Techniques:

Separate chaining:-

- which introduces an additional field with data. i.e chain. A separate chain table is maintained for collision data when collision occurs then a linked list (chain) is maintained at the home bucket

Ex:- $H(key) = key \% D$

whele D is the size of table. D=10



Open Addressing - (Linear Probing):-

~~Th~~ This when collision occurs i.e. when a 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, the hash table is represented as a 1-dimensional array with indices that range from 0 to the desired table size-1.
- Before incerting any elements into this table, we must initialize the table to represent the situation where all slots are empty.
- This allows us to detect overflow and collisions when we insert elements into the table. then using some ~~situat~~ suitable hash function the element can be inserted into the hash table.

Ex:-

H(Key) = Key % table size
= Key % 10

31, 4, 8, 7,

0	NULL
1	31
2	NULL
3	NULL
4	4
5	NULL
6	NULL
7	7
8	8
9	NULL

— Next Key is 21

H(Key) = 21 % 10 = 1

— So, index 1 is already occupied by 31. To resolve this collision we will Linearly move down and at the next empty location we will Prob the element

∴ 21 will be Placed at the index 2

0	null
1	31
2	21
3	NULL
4	4
5	null
6	null
7	7
8	8
9	NULL

insert 5, 31, 61, 99

0	
1	31
2	21
3	31
4	4
5	5
6	61
7	7
8	8
9	9

⇒ now insert 99, index 9 is alread filled with 9 so, there is no next empty bucket. so, overflow will occurs.

⇒ To handle it we move back to bucket 0 and is the location over there is empty 99 will be placed at 0th index.

Problem with linear Probing

- One major Problem with linear Probing is Primary clustering. Primary clustering is a ~~Problem~~ Process in which a block of data is formed in the hash table when collision is resolved.

Ex:-

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

0	39
1	29
2	8
3	
4	
5	
6	
7	
8	18
9	19

cluster is formed

Rest of the table is empty

- This cluster Problem can be solved by quadratic Probing.

Quadratic Probing:-

- Quadratic Probing operates by taking the original hash value and adding successive values of an arbitrary quadratic Polynomial to the starting value.

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

- where m can be a table size (or) any Prime number

Ex:-

37, 90, 55, 22, 11, 17, 49, 87

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

0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	17
9	49

- $17 \% 10 = 7$ Collision will occur

- So, we will apply quadratic Probing to insert this record in the hash table

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

$i=0 \quad (17+0^2) \% 10 = 7$ – Collision

$i=1 \quad (17+1^2) \% 10 = 8$ – bucket is empty, then insert

- $49 \% 10 = 9$ – bucket is empty, then insert

- $87 \% 10 = 7$ – Collision

Apply ~~Linear~~ Quadratic Probing

$i=0 \quad (87+0) \% 10 = 7$ – Collision

$i=1 \quad (87+1) \% 10 = 8$ – Collision

$i=2 \quad (87+2^2) \% 10 = 1$ – Collision

$i=3 \quad (87+3^2) \% 10 = 6$ – ~~collision~~ bucket is free so insert at indx 6

Double Hashing :-

- Double hashing is technique in which a second hash function is applied to the key when a collision occurs.

- By applying the 2nd hash function we will get the number of Positions from the Point of Collision to insert

- there ale 2 important rules to be followed for the 2nd function:
  1. it must never evaluate to zero
  2. Must make sure that all Cells can be Probed.

Formula :

$$H_1(\text{key}) = (\text{key}) \bmod (\text{table size})$$

$$H_2(\text{key}) = M - ((\text{key}) \bmod M)$$

– Whele M is a Prime number and it is smaller than the size of the table

Ex:- 37, 90, 45, 22, 17, 49, 55

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

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

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

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

$H_1(17) = 17 \% 10 = \underline{7}$ Collision

$\Rightarrow H_2(\text{Key}) = M - (\text{Key} \% M)$

$\therefore M = 7$

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

$= 7 - 3 = 4$

→ Now insert 17 at 4 places from 7 index i.e index 1

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

$H_1(49) = 49 \% 10 = 9$

$H_1(55) = 55 \% 10 = \underline{5}$ collision

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

$= 7 - 6 = 1$

→ Now insert 55 at 1 places from 5 index i.e index 6

Rehashing :-

- in which the table is resized. i.e the size of table is doubled by creating a new table. it is Preferable if the total size of table is a Prime number.

  - Rehashing required situations ale:

    1. When table is completely full
    2. With quadratic Probing when the table is filled half
    3. When insertions fail due to overflow

- In such situations, we have to transfer entries from old table to the new table by recomputing their Positions using suitable Hash functions.

Ex:-

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

H(key) = (key) mod (Table size)

37 % 10 = 7

90 % 10 = 0

55 % 10 = 5

22 % 10 = 2

17 % 10 = 7 Collision

solved by Linear Probing i.e index 8

49 % 10 = 9

87 % 10 = 7

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

- Now this table is almost full and if we try to insert more elements collisions will occur and 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 Prime, we will ~~make~~ Perform to make the table size as 23.

H(key) = key % 23

27 % 23 = 14

90 % 23 = 21

55 % 23 = 9

22 % 23 = 22

17 % 23 = 17

49 % 23 = 3

87 % 23 = 18

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

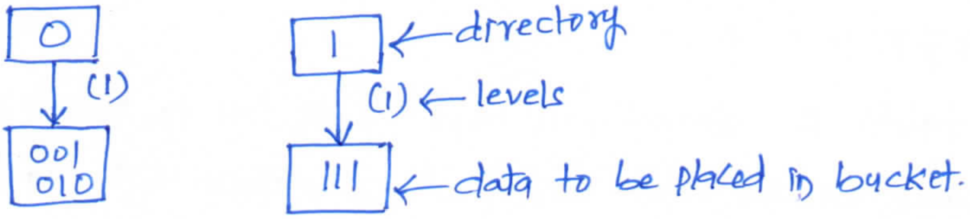
Extensible Hashing:-

— Extensible hashing 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 grows and shrinks similar to B-Trees.

- it referring the size of directory the elements are to be placed in buckets. the levels are indicated in Parenthesis.

syntax:

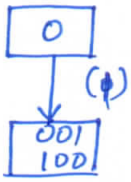


— The bucket can hold the data of its global depth. if data in bucket is more than global depth then split the bucket and double the directory.

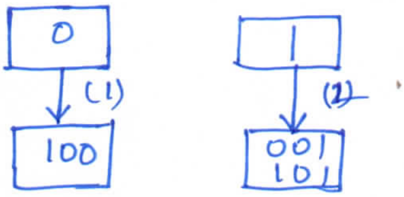
Ex: 1 4 5 7 8 10

step①: insert 1, 4

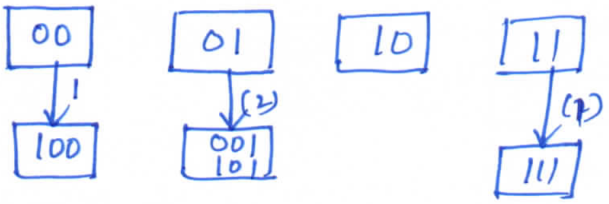
1=001
4=100



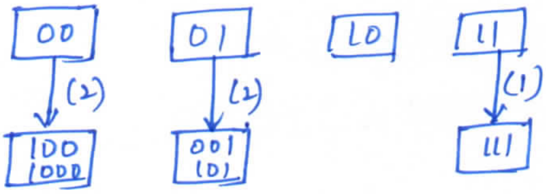
- insert 5: 101
The bucket is full hence double the directory



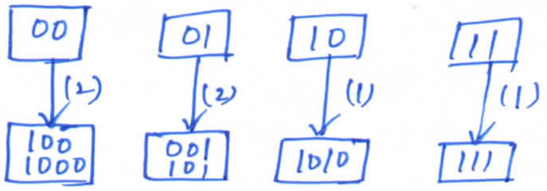
step② insert 7: 111
depth is full then double the directory and split bucket



step③ insert 8: 1000



step④ 10: 1010


00
(2)
100
1000
01
(2)
001
101
10
(1)
1010
11
(1)
111