UNIT-V

SEARCHING, SORTING AND HASHING TECHNIQUES.

1) Searching
2) Linear Search
3) Binary Search
4) Sorting
5) Bubble Sort
6) Selection Sort
7) Insertion Sort
8) Shell Sort
9) Radix Sort
10) Hashing
11) Hash functions.
12) Separate chaining.
13) Open Addressing.
14) Rehashing.
15) Extendible Hashing.

Searching: It is an operation (or) a technique that helps finds the place of a given element (or) value in the list. Any search is said to be successful (or) Unsuccessful depending upon whether the element that is being searched is found (or) not. Some of the standard searching technique that is being followed in the data structure is listed below.

* Linear Search (or) Sequential search. } N/D - 2018
* Binary Search.

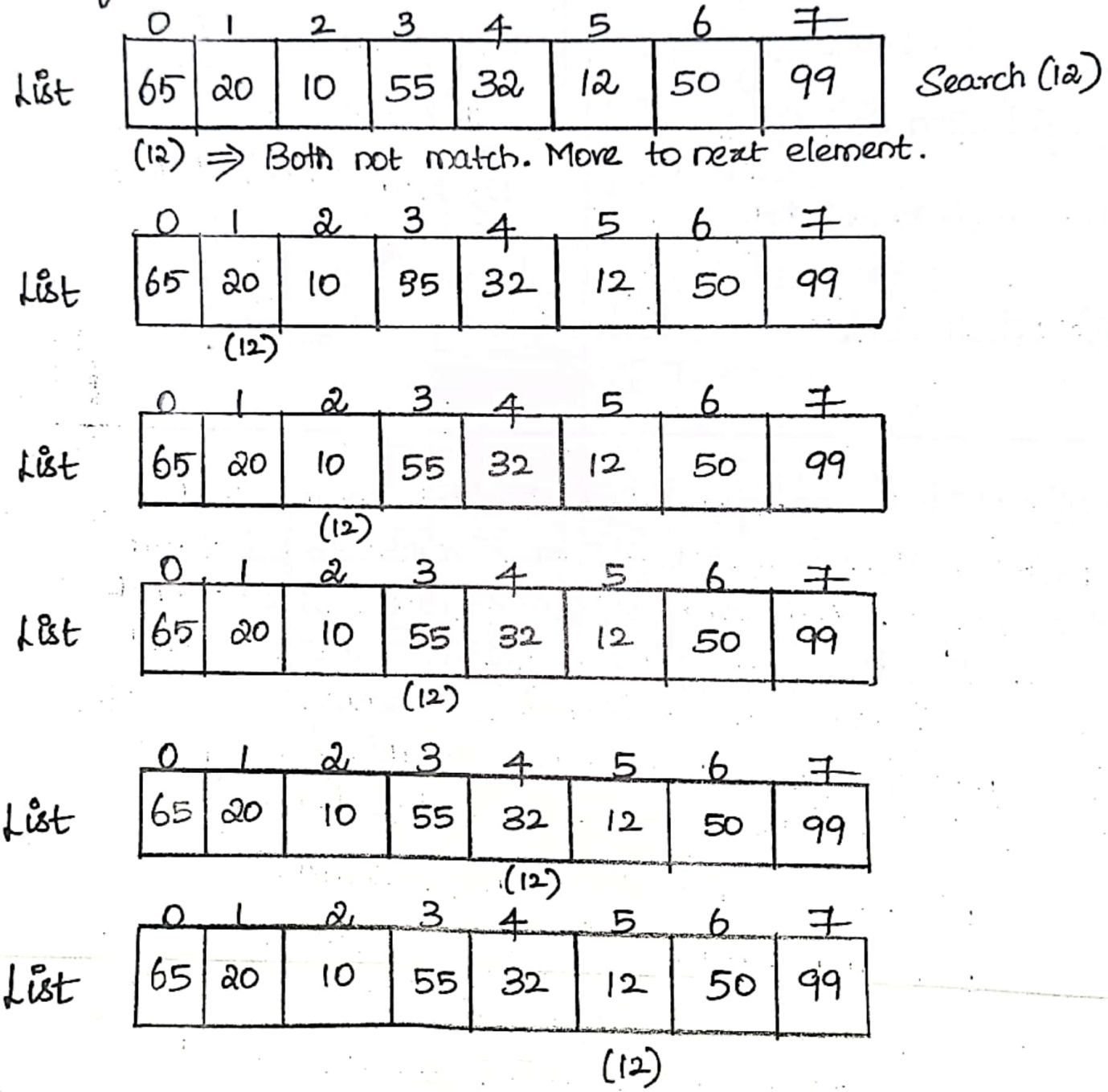
Linear Search (or) Sequential Search:

It is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item

is returned, Otherwise the search continues till the end of the data collection.

Steps:

1) Start from the leftmost element of arr[] and one by one compare x with each element of arr[].
2) If x matches with an element, return the index.
3) If x doesn't match with any of elements, return -1.



Both are matching. So we stop comparing and display element found at Index 5.

arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}

x = 175.

Output :- -1.

Element x is not present in arr[], so it returns -1.

Assume that K is an array of 'n' keys, K(0) through K(n-1), and r an array of records, r(0) through r(n-1), such that K(i) is the key of r(i).

```
for (i=0; i<n; i++)
  if (key == K(i))
    return (i);
  else
    return (-1);
```

If match found then index value is returned.
If no match found then, -1 is returned.

Advantages:

1) The Linear search is simple. It is very easy to understand and implement.

2) It does not require the data in the array to be stored in any particular order.

DISADVANTAGES:

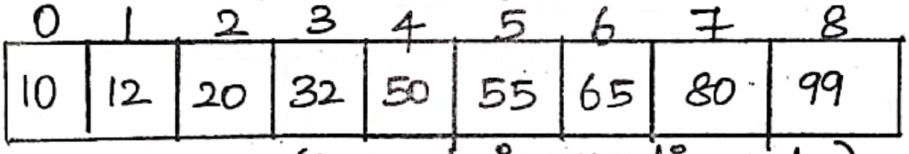
1) This method is insufficient, when large number of elements is present in list.

2) It consumes more time and reduces the retrieval rate of the system.

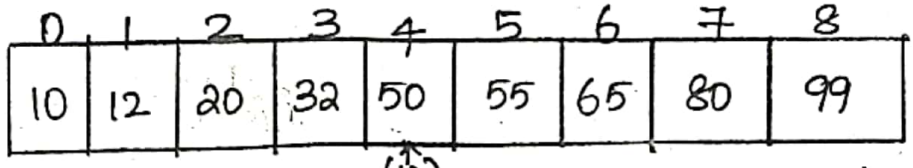
(Eg) The Linear search is inefficient - If array being searched contains 20,000 elements, the algorithm will have to look at all 80,000 elements in order, to find a value in the last element.

(3) BINARY SEARCH

Binary Search, is also known as half-interval Search, Logarithmic search, (or) binary chop, is a search algorithm that finds the position of a target value within a sorted array. It compares the target value to the middle element of the array. I/P: 65, 20, 10, 55, 32, 12, 50, 99



(Arranged in ascending order)

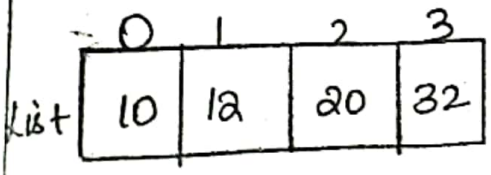


8/2 = 4

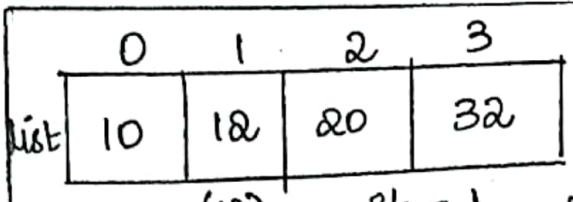
7/2 = 3

Search element (12), compared with middle element (50).

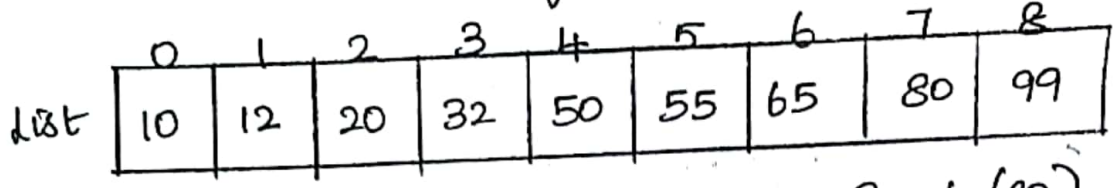
12 < 50. So search only in the left sublist (10, 12, 20, 32)



5/2 = 2

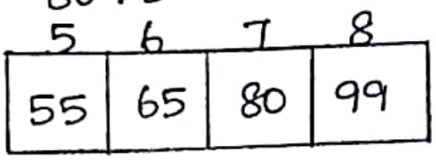


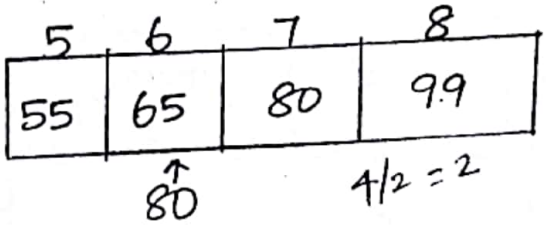
Search element (12), Compared with middle element (12). Both are matching. So result is "Element found at Index. 1".

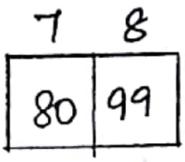


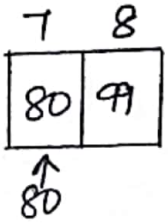
Search (80)

80 > 50









Both are matching. So element 80 is found at Index 7.

Algorithm:

```
low = 0;
hi = n-1;
while (low <= hi)
{
⇒ mid = (low + hi)/2;
if (key == k(mid))
    return (mid);
if (key < k(mid))
    hi = mid-1;
else    low = mid+1;
} /* end while */
return (-1);
```

Advantages:

(1) faster, because does not have to look at every element.

(2) Disadvantages:

(1) This algorithm requires the list to be sorted. Then only the method is applicable.

(4) SORTING

Sorting refers to arranging data in a particular format. It specifies the way to arrange data in particular order.

Eg; Telephone directory - The telephone directory stores the telephone numbers of people sorted by their name, so it is easily searched by their names.

Some of the Sorting techniques are.,

1) Bubble Sort
2) Selection Sort
3) Insertion Sort
4) Shell Sort
5) Radix Sort.

BUBBLE SORT

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison - based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in Order.

ALGORITHM :

We assume list is an array of 'n' elements. We further assume that swap function swaps the values of the given array elements.

```
begin Bubble Sort (list)
for all elements of list
 if list [i] > list [i+1]
 swap ( list [i], list [i+1])
   end if
   end for
 return list
 end bubble_sort.
```

(Eg) 14, 33, 27, 35, 10.

Bubble sort starts with very first two elements, Comparing them to check which one is greater.

14, 33, 27, 35, 10. [14 < 33 Already Sorted no swap]

14, 33, 27, 35, 10. [33 > 27. So swap it]

14, 27. 33, 35, 10 [33 < 35. Already Sorted]

14, 27, 33, 35, 10 [35 > 10. so swap it]

14, 27, 33, 10, 35 [33 > 10. so swap it]

14, 27, 10, 33, 35 [27 > 10. so swap it]

14, 10, 27, 33, 35 [14 > 10. so swap it]

10, 14, 27, 33, 35 [Last Iteration]

And when there is no swap required, bubble sort learns that an array is completely sorted.

SELECTION SORT

Selection sort is a simple algorithm. This sorting algorithm is in-place comparison based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted Path at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

14	33	27	10	35	19	42	44

The whole list is scanned sequentially.

14 33 27 10 35 19 42 44 (swap)
↑min

So, we replace 14 with 10. 10 is the minimum value in the list, appears in the 1st position of the sorted list.

10, | 33 27 14 35 19 42 44
(Sorted) | (unsorted)
MIN

We find that 14 is the second lowest value in the list and it appear at the 2nd place. Swap the values.

10, 33, 27, 14, 35, 19, 42, 44
(After swapping two least values are positioned)

10, 14, | 27, 33, 35, 19, 42, 44 (swap it)
SORTED | UNSORTED
MIN

10, 14, 19, | 33, 35, 27, 42, 44 (swap it)
SORTED UNSORTED MIN

10, 14, 19, 27, | 35, 33, 42, 44 (swap it)
SORTED UNSORTED MIN

10, 14, 19, 27, 33, | 35, 42, 44 (swapping No Need)
SORTED UNSORTED MIN

10, 14, 19, 27, 33, 35, | 42, 44 (No Need swapping)
SORTED UNSORTED MIN

So the Sorted Order is

10, 14, 19, 27, 33, 35, 42, | 44.
SORTED (UNSORTED) MIN

Solution: 10, 14, 19, 27, 33, 35, 42, 44 [Sorted array]

Eg; 7 5 4 2

(2)

2 ⇒ 5 4 7

min element

Sorted array Unsorted Array

2 5 4 7

2 ⇒ 4 5 7

min element

Sorted array Unsorted array

2 4 5 7 ⇒ 2 4 5 7

min element

Sorted array Unsorted array

2 4 5 7

min element ⇒ 2 4 5 7

Sorted array.

ALGORITHM:

```
Void Selection_Sort (int arr[], int n)
{ int i, j, min_idx;
for (i=0; i<=n-1; i++)
  {
```

```
        min_idx = i;
    for (j = i+1; j < n; j++)
    if (arr[j] < arr[min_idx])
        min_idx = j;
    swap(&arr[min_idx], &arr[i]);
    }
}
```

INSERTION SORT

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. for eg., the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted here. Hence the name, insertion sort.

Steps:

1) If it is the 1st element, it is already sorted. Return 1.
2) Pick next element.
3) Compare with all elements in the sorted sub-list.

4) Shift all the elements in the sorted sub-list that is greater than the value to be sorted.

5) Insert the value.

6) Repeat until list is sorted.

Eg (1)

							Position Moved
Original	34	8	64	51	32	21	
(Pass)							
After P=1	8	34	64	51	32	21	1
After P=2	8	34	64	51	32	21	0
After P=3	8	34	51	64	32	21	1
After P=4	8	34	51	64	21	32	3
	8	34	51	21	64	32	
	8	34	21	51	64	32	
	8	21	34	51	64	32	
After P=5	8	34	51	32	64	21	3
	8	34	32	51	64	21	
	8	32	34	51	64	21	
After P=5	8	32	34	51	21	64	4
	8	32	34	21	51	64	
	8	32	21	34	51	64	
	8	21	32	34	51	64	

No. of elements : 6.

n-1 : 6-1 => 5 passes.

ALGORITHM :

```
Void InsertionSort ( ElementType A[], int N)
{
    int j, P;
```

```
Element Type Tmp;
 for (P=1; P<N; P++)
{
     Tmp = A[P];
   for (J=P; J>0 && A[J-1] > Tmp; J--)
    A[J] = A[J-1];
    A[J] = Tmp;
  }
}
```

Eg(a): No. of elements : 8
(n-1) : 8-1 ⇒ 7 passes.

		Position Moved
Original	14 33 27 10 35 19 42 44	
After P=1	14 33 27 10 35 19 42 44	0
After P=2	14 27 33 10 35 19 42 44	1
After P=3	14 27 10 33 35 19 42 44 14 10 27 33 35 19 42 44 10 14 27 33 35 19 42 44	3
After P=4	10 14 27 33 35 19 42 44	0
After P=5	10 14 27 33 19 35 42 44 10 14 27 19 33 35 42 44 10 14 19 27 33 35 42 44	3
After P=6	10 14 19 27 33 35 42 44	0
After P=7	10 14 19 27 33 35 42 44	0

SHELL SORT N/D-2018

Shell sort named after its inventor, Donald Shell, also referred as diminishing increment sort.

Algorithm:

```
Void
Shellsort (ElementType A[], int N)
{
    int i, j, Increment;
    ElementType Tmp;
for ( Increment = N/2 ; Increment >0 ; Increment /=2)
for ( i= Increment ; i< N ; i++)
{
    Tmp = A[i];
 for (j=i ; j>= Increment ; j- = Increment)
    if ( Tmp < A [j- Increment])
    A[jJ = A [j- Increment];
   else
     break;
   A[jJ = Tmp;
  }
}
```

compare elements that are distant apart rather than adjacent. so if there are N elements then we start to solve gap <N.

gap = floor (N/2)

⇒ gap1 = floor (gap/2)

gap2 = floor (gap1/2)

Eg; Original list : 77 62 14 9 30 21 80 25 70 55

N = 10.

gap = floor (N/2) = floor (10/2) = 5.

Pass1 : 77 62 14 9 30 21 80 25 70 55

So checking the elements distance apart.

⇒ 21 62 14 9 30 77 80 25 70 55

(77, 21) ⇒ swap

(62, 80) (14, 25), (9, 70) (30, 55) ⇒ No swap.

Pass 2 : Gap = floor (5/2) = 2.

⇒ 21 62 14 9 30 77 80 25 70 55

⇒ 14 9 21 25 30 55 70 62 80 77

Pass 3 : Gap = floor (2/2) = 1.

⇒ 14 9 21 25 30 55 70 62 80 77

⇒ 9 14 21 25 30 55 62 70 77 80

(swap) (swap) (swap)

Sorted List :-

9 14 21 25 30 55 62 70 77 80

RADIX SORT

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits, which share the same significant position and value.

Radix sort
- Least significant Digit (LSD)
- Most Significant Digit (MSD)

LSD → Process the integer representations starting from least digit and move towards the MSD.

MSD → Process the integer representations starting from maximum digit and move towards the LSD.

```
radix sort (inout the Array : Item Array,
            in n : integer , in d : integer)
// Sort n d-digit integers in the array the
  for (j=d down to 1)
{
   Initialize 10 groups to empty
   Initialize a counter for each group to 0.
   for (i=0 through n-1)
   {  k = jth digit of the Array[i]
      Place the Array[i] at the end of group k
      Increase kth counter by 1
   }
```

Replace the items in the Array with all the items in group 0, followed by all the items in group 1, and so on.

Eg (1) 170 45 75 90 802 24 2 66

170 045 075 090 802 024 002 066

Consider one's place.

170 090 802 002 024 045 075 066

Consider 10th place.

802 002 024 045 066 170 075 090

Consider 100th place.

002 024 045 066 075 090 170 802

Sorted array: 2 24 45 66 75 90 170 802.

Eg (2): 53 89 150 36 633 233 733.

053 089 150 036 633 233 733

Consider one's place.

150 053 633 233 733 036 089

Consider 10th place.

633 233 733 036 150 053 089

Consider 100th place

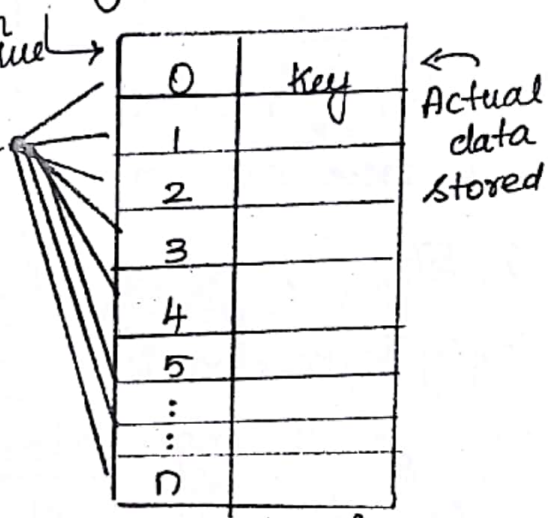
036 053 089 150 233 633 733

Sorted array: 36 53 89 150 233 633 733

HASHING

Hashing is the process of mapping large amount of data item to a smaller table with the help of a hashing function. The essence of hashing is to facilitate the next level searching method when compared with the linear (or) Binary search.

It is the process of indexing and retrieving element (data) in a datastructure to provide faster way of finding the element using the hash key.



Key → Hash function → Hash value

↑ "Actual data to be stored"

"Hash table" is just an array which maps a key (data) into the datastructure with the help of hash function such that insertion, deletion and search operations are performed with constant time complexity.

"Hash function" is a function which takes a pice of data (ie., key) as input and outputs an integer (ie., hash value) which maps the data to a particular index in the hash table.

Every entry in the hash table is based on the key value generated using a hash function. The values returned by a hash function is called hash values, hash codes, digests (or) simply hashes.

HASH FUNCTION.

The mapping between an item and the slot where that item belongs in the hash table is called the hash function. The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and m-1.

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

Shows the hash table of size m=11. In other words, there are m slots in the table, from 0 to 10.

Eg; 54, 26, 93, 17, 77 and 31.

Simply takes an item and divides it by the table size, returning the remainder as its hash value.

h(item) = item % m

Item	Hash value	
54	10	(54 mod 11)
26	4	(26 mod 11)
93	5	(93 mod 11)
17	6	(17 mod 11)
77	0	(77 mod 11)
31	9	(31 mod 11)

11) 54 (4
44
10

11) 26 (2
22
4

Once the hash values have been computed, we can insert each item into the hash table at the

designated position.

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Note that 6 out of 11 slots are occupied. This is referred to as "LOAD FACTOR", and commonly denoted by $\lambda = \frac{\text{number of items}}{\text{Table size}}$.

Eg., $\lambda = 6/11$.

for eg., if the item 44 had been the next item in our collection, it would have a hash value $(44 \bmod 11) = 0$. since 77 also had a hash value of 0, we would have a problem.

According to the hash function, two (or) more items would need to be in the same slot. This is referred to as a "collision" also called as "clash". So collision create a problem in hashing technique.

Two methods of Hash functions.

* folding method.
* mid-square method.

folding method:

If our item was the phone number

436-555-4601.

(43, 65, 55, 46, 01) groups of 2.

$43+65+55+46+01 \Rightarrow 210$.

If hash table has 11 slots, then $(210 \bmod 11) = 1$.

So, the phone number 436-555-4601 hashes to slot 1.

Some folding methods go one step further and reverse every other piece before the addition.

Eg; 436-555-4601

Group of 2 (43, 65, 55, 46, 01)

Reverse ⇒ 43, 56, 55, 64, 01

(43 = No change; 56, 55, 64 → Reverse; 01 = No change)

$43 + 56 + 55 + 64 + 01 = 219.$

$(219 \bmod 11) = 10.$

MID-SQUARE METHOD :

Another numerical technique for constructing a hash function is called the mid-square method. We first square the item, and then extract some portion of the resulting digits.

If the item were 44, then $44^2 = 1936$.

By extracting the middle two digits, 93 and performing the remainder step we get $(93 \bmod 11) = 5$.

Item	Remainder	Mid-Square
54	10	3
26	4	7
93	5	9
17	6	8
77	0	4
31	9	6

We can also create hash functions for character-based items such as strings. The word "cat" can be thought of as a sequence of ordinal values.

Ord ('c') = 99
Ord ('a') = 97
Ord ('t') = 116

a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107
l 108	m 109	n 110	o 111	p 112	q 113	r 114	s 115	t 116	u 117	v 118
w 119	x 120	y 121	z 122.							

I/p String : cat (99 + 97 + 116) = 312

(312 mod 11) = 4.

(or)

To use positional value as a weighting factor.

Position :

1 2 3

C A T

(99x1)+(97x2)+(116x3) = 641 (641 mod 11) = 3.

SEPARATE CHAINING (N/D – 2018)

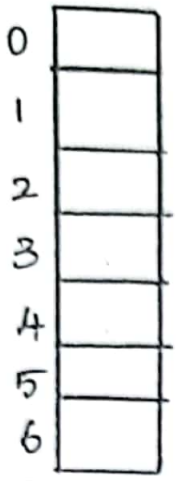
The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as "key mod 7" and sequence of

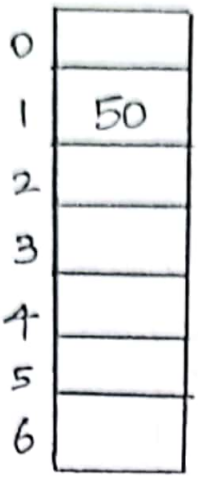
Keys as

50, 700, 76, 85, 92, 73, 101.

Eg (1)



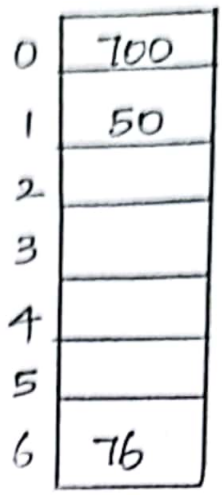
Initial Empty Table



Insert 50

(50 mod 7)

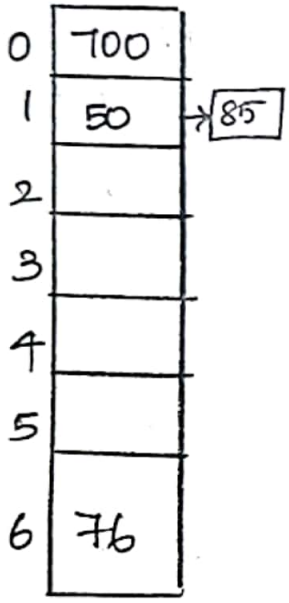
= 1



Insert 700 and 76

(700 mod 7) (76 mod 7)

= 0 = 6

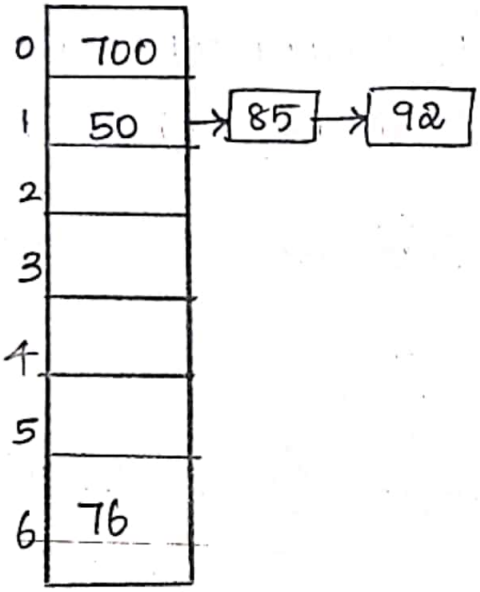


Insert 85

(85 mod 7)

= 1

Collision occurs, add to chain

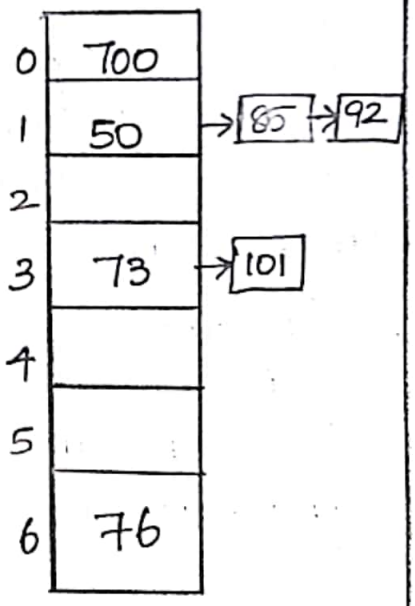


Insert 92

(92 mod 7)

= 1

Collision occurs, add to chain



Insert 73 and 101

(73 mod 7)

= 3

(101 mod 7)

= 3

Collision occurs, add to chain.

OPEN ADDRESSING (N/D-2018)

Open addressing (or) closed hashing is a method of collision resolution in hash tables.

formula : $h_0(key) = (key \bmod arraysize)$.

With this method a hash collision is resolved by Probing (or) searching through alternate locations in the array, until either the target record is found, (or) an unused array slot is found, which indicates that there is no such key in the table. Well known probe sequence includes;

- → Linear probing (N/D-2018)
- → Quadratic probing
- → Double Hashing.

Operations : Insert (key)
Search (key)
Delete (key)

Linear probing :

In which the interval between probes is fixed - often set to 1.

Quadratic probing :

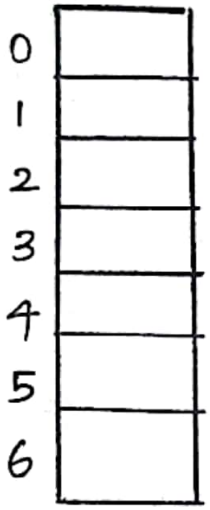
In which the interval between probes increases linearly (hence, the indices are described by a Quadratic function).

Double Hashing : In which the interval between probes is fixed for each record but is computed by another hash function.

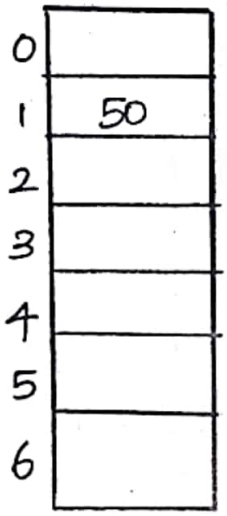
LINEAR PROBING.

Eg ; Let us consider a simple hash function as "Key mod 7" and sequence of keys as.

50, 700, 76, 85, 92, 73 and 101.

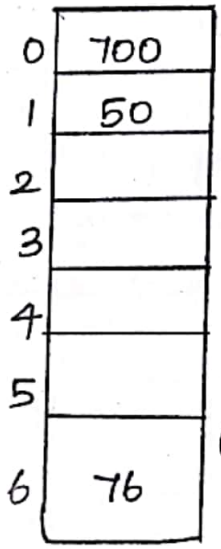


Initial Empty Table



(50 mod 7) = 1

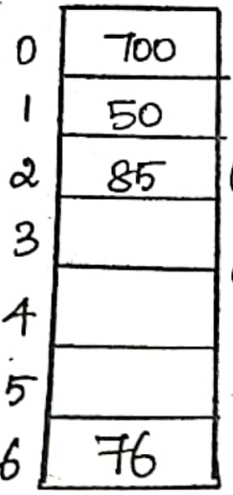
Insert 50



(700 mod 7) = 0

(76 mod 7) = 6

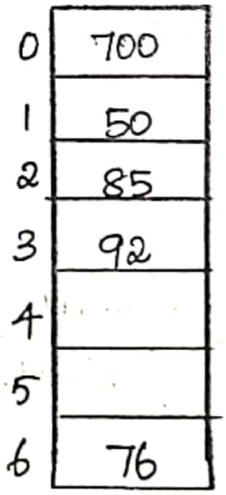
Insert 700 and 76



(85 mod 7) = 1 collision occurs

Insert 85.

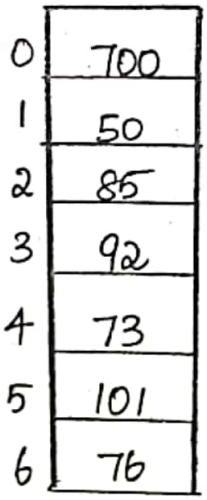
Collision occurs so insert 85 at next free slot.



(92 mod 7) = 1

Insert 92.

Collision occurs as 50 is there at index 1. so insert at next free slot.



(73 mod 7) = 3

(101 mod 7) = 3.

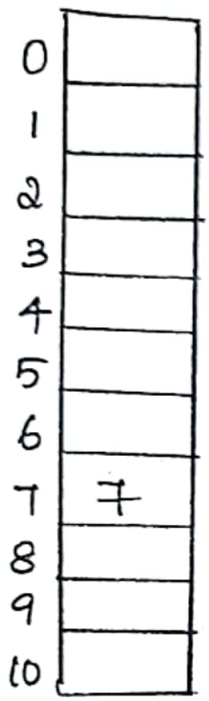
Insert 73 and 101.

Collision occurs. so insert at next free slot.

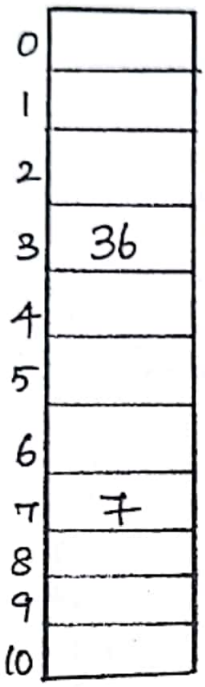
QUADRATIC PROBING

$h_0(x) = (\text{Hash}(x) + i \wedge 2) \,\%\, \text{Hash Table size}$

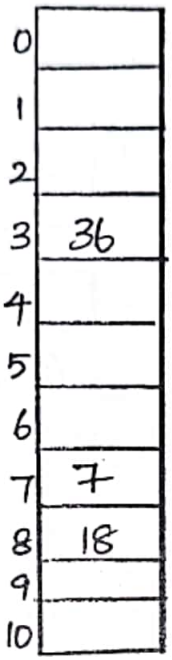
Eg; Keys : 7, 36, 18, 62. and use Hash table size as 11.



Insert 7.
(7 mod 11)
= 7

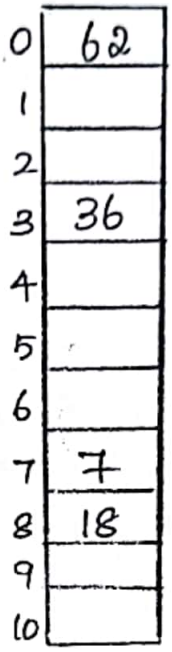


Insert 36.
(36 mod 11)
= 3.



Insert 18
(18 mod 11)
= 7.
Collision occurs.

$h_1(18) = [(18 + 1*1) \bmod 11]$
$= [19 \bmod 11]$
$= 8$



Insert 62.
(62 mod 11)
= 7
Collision occurs

$h_{62} = [(62 + 2*2) \bmod 11]$
$\Rightarrow [66 \bmod 11]$
$\Rightarrow 0.$

DOUBLE HASHING.

$h_i(x) = (\text{Hash}(x) + i * \text{Hash } 2(x)) \,\%\, \text{HashTablesize}$

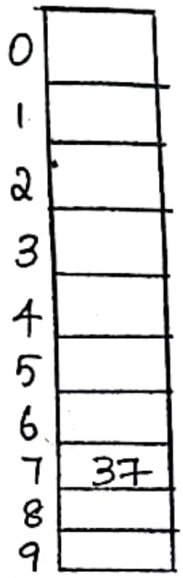
$H_1(\text{key}) = \text{key mod tablesize.}$

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

M is the prime number < table size

Eg., 37, 90, 45, 22, 17, 49, 55

Table size = 10.



Insert 37

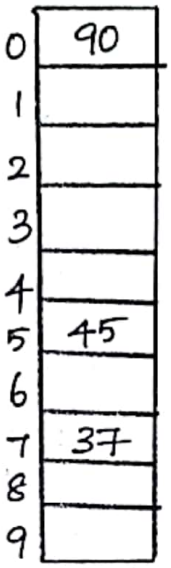
(37 mod 10)

= 7

Insert 90

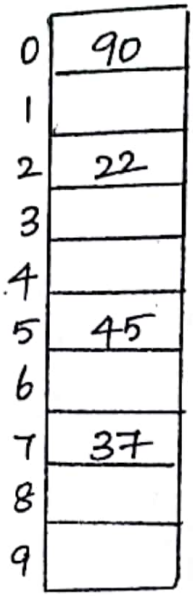
(90 mod 10)

= 0



(45 mod 10)

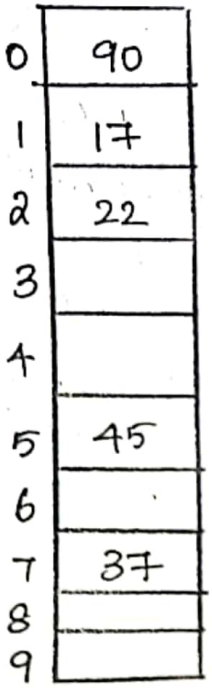
= 5



Insert 22

(22 mod 10)

= 2



Insert 17

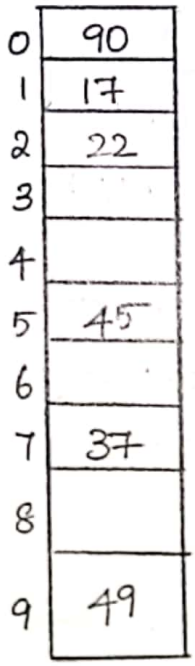
(17 mod 10) = 7

$H_2(17) = 7 - (17 \bmod 7)$ (Prime no. less than 10)

$= 7 - 3$

$= 4.$

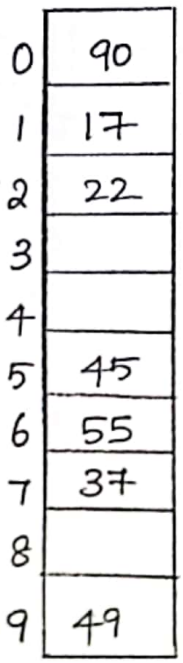
Place 17 in such a way that (4) position from (7)



Insert 49

(49 mod 10)

= 9



Insert 55

(55 mod 10)

= 5.

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

$= 7 - 6$

$= 1.$

REHASHING. (N/D-2018)

Rehashing is the process of re-calculating the hashcode of already stored entries (key value pairs) to move item to another bigger size Hashmap when load factor threshold is reached.

LOAD FACTOR : It is a measure, "till what load, hashmap can allow elements to put in it before its size is increased.

When the number of item in map, crosses the load factor limit at that time Hashmap double its capacity and hashcode is re-calculated of already stored elements for even distribution of key value pairs accross new buckets.

Steps :

1) create a Large table.

2) create a new hash function (for example, the table size has changed)

3) Use the new hash function to add the existing data items from the old table to the new table.

Rehashing Techniques :

1) Linear probing
2) Two-pass file creation
3) Separate overflow area
4) Double Hashing
5) Synonym chaining
6) Bucket Addressing
7) Bucket chaining.

Eg; 13, 15, 24 and 6 are inserted into an open addressing hash table of size 7.

hash function $h(x) = x \bmod 7$.

Linear probing with i/p 13, 15, 24 and 6.

0	6
1	15
2	
3	24
4	
5	
6	13

$(13 \bmod 7) = 6$

$(15 \bmod 7) = 1$

$(24 \bmod 7) = 3$

$(6 \bmod 7) = 0$

After 23 is inserted.

0	6
1	15
2	23
3	24
4	
5	
6	13

$(23 \bmod 7) = 2$

∴ Table is 70% full. So a new table is created. The size of the table is 17, because this is the 1st prime number that is twice as large as the old table size.

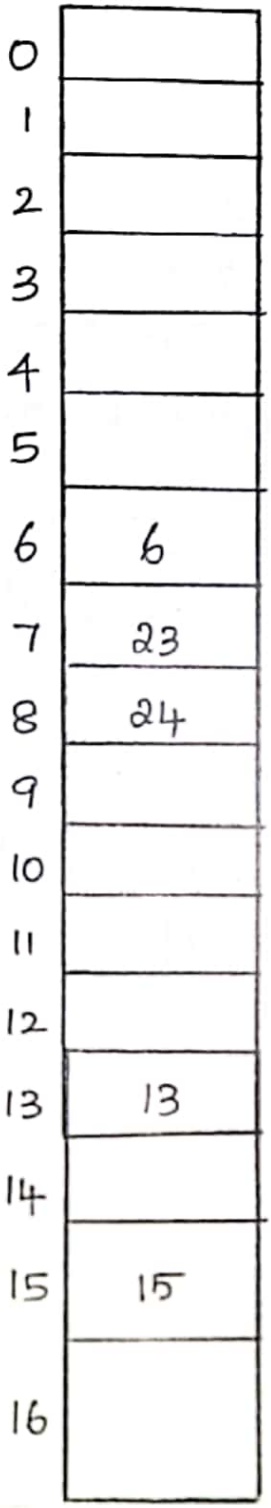
∴ New hash function is then.,

$h(x) = x \bmod 17$.

The old table is scanned, and elements 6, 15, 23, 24 and 13 are inserted into the new table. This entire operation is called as rehashing.

Open addressing hash table after rehashing.

6, 15, 23, 24 and 13.



Good.

Rehashing is expensive operation with running time O(N).

Extendible Hashing.

Used when the amount of data is too large to fit in main memory and external storage is used.

N records in total to store.

M records in one disk block.

In ordinary hashing several disk blocks are examined, to find an element, a time consuming process.

Extendible hashing : No more than two blocks are examined.

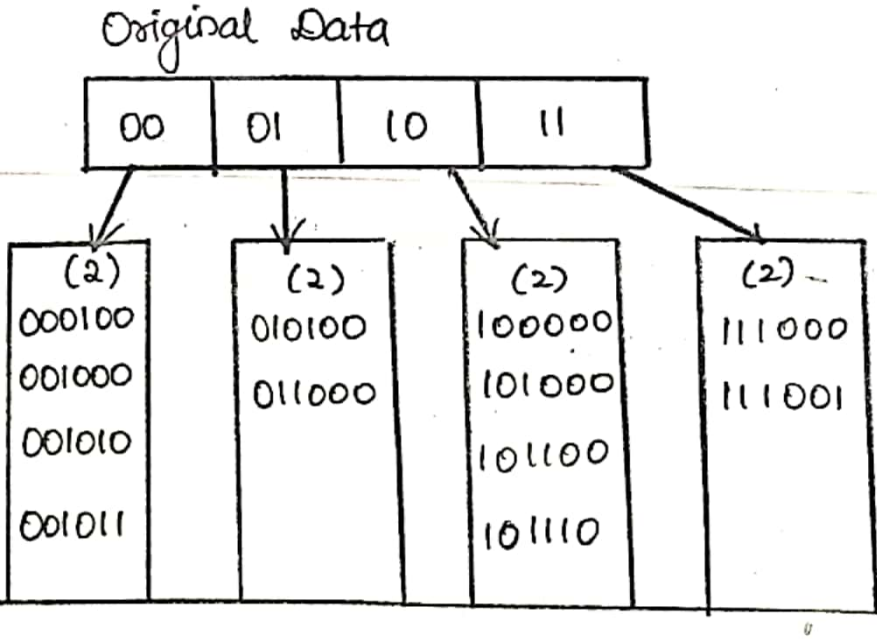
Idea :

→ Keys are grouped according to the first m bits in their code.

→ Each group is stored in one disk block.

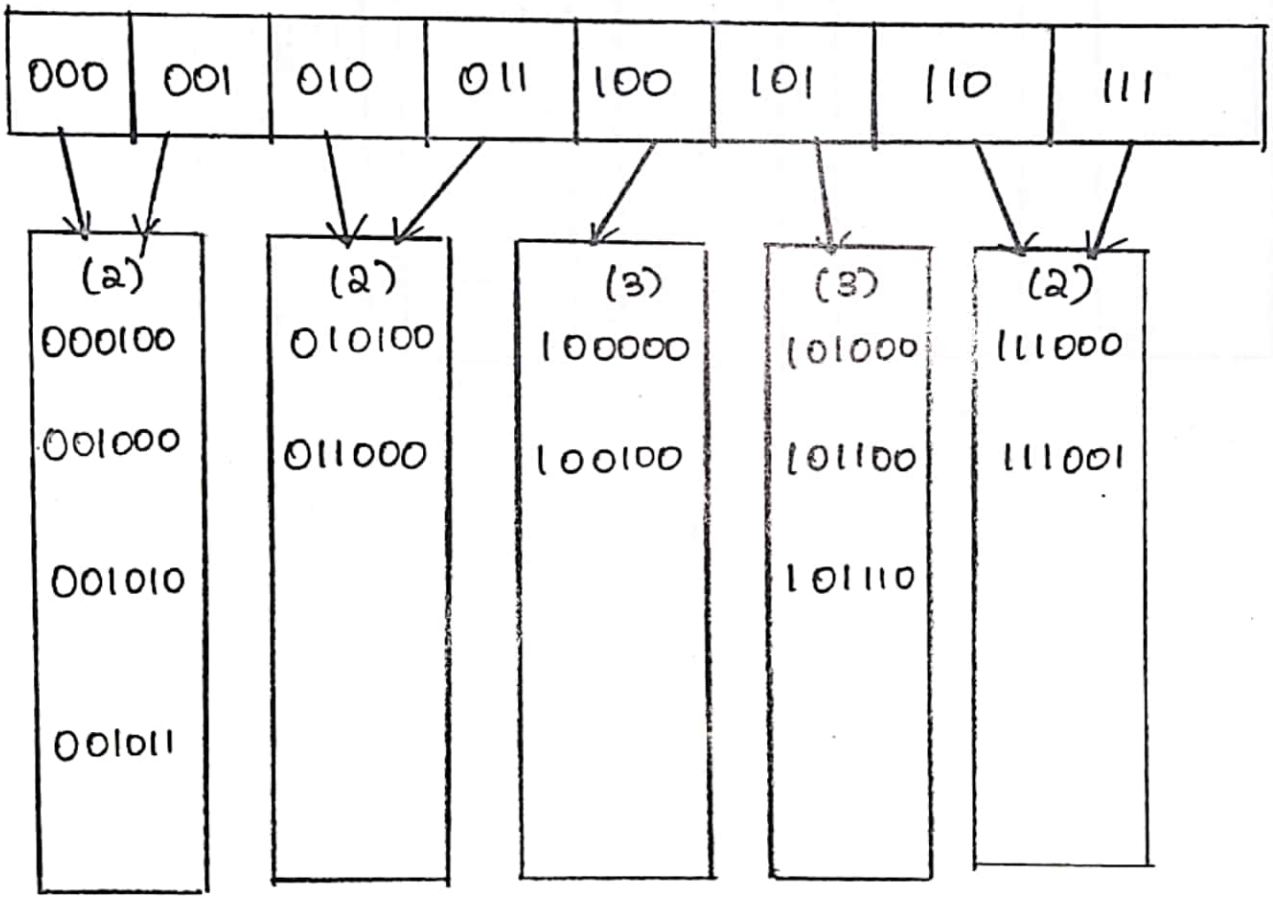
→ If the block becomes full and no more records can be inserted, each group is split into two, and m+1 bits are considered to determine the location of a record.

Eg., Suppose data consists of several six-bit integers. Each leaf has up to M = 4 elements.



D → Directory (no. of bits used by the root).
The no. of entries in the directory is thus 2^D.

Suppose we need to insert 100100. This would go into third leaf, but it is already full, there is no room. We thus split this leaf into two leaves, which are now determined by the first three bits. This requires increasing the directory size to 3.



Extendible Hashing after insertion of 100100 and directory split.

Extendible hashing after inserting of 000000 and leaf split.