UNIT 6

SEARCHING

Searching 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 depUBing 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
- Binary Search

Linear Search:

This is the simplest method for searching. In this technique of searching, the element to be found in searching the elements to be found is searched sequentially in the list. This method can be performed on a sorted or an unsorted list (usually arrays). In case of a sorted list searching starts from 0^{th} element and continues until the element is found from the list or the element whose value is greater than (assuming the list is sorted in ascUBing order), the value being searched is reached.

As against this, searching in case of unsorted list also LBins from the 0^{th} element and continues until the element or the UB of the list is reached.

One-Dimensional Array having 7 Elements

Example:

The list given below is the list of elements in an unsorted array. The array contains ten elements. Suppose the element to be searched is '46', so 46 is compared with all the elements starting from the 0^{th} element, and the searching process UBs where 46 is found, or the list UBs.

The performance of the linear search can be measured by counting the comparisons done to find out an element. The number of comparison is 0(n).

```
Linear Search ( Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Complexity of algorithm

Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)

C Program

```
#include<stdio.h>
#include<conio.h>
void main ()
{
   int a[10] = {10, 23, 40, 1, 2, 0, 14, 13, 50, 9};
   int item, i,flag;
   printf("\nEnter Item which is to be searched\n");
   scanf("%d",&item);
   for (i = 0; i< 10; i++)
   {
      if(a[i] == item)
      {
       flag = i+1;
          break;
      }
```

```
      else
      flag = 0;
   }
\\   if(flag != 0)
   {
      printf("\nItem found at location %d\n",flag);
   }
   else
   {
      printf("\nItem not found\n");
   }
}
```

Output:

```
Enter Item which is to be searched
20
Item not found
Enter Item which is to be searched
23
Item found at location 2
```

Binary search:

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer.

For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the sub array reduces to zero.

Binary search is implemented using following steps...

Step 1 – initialize the **low** = lower bound(0) and **high** = upper bound**(n-1)**

Read the search element from the user.

Step 2 - Find the middle element in the sorted list.

Mid= (Low + High)/2

Step 3 - Compare the search element with the middle element in the sorted list.

Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function. **Key == a[Mid]**

Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.

Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

Key < a[Mid]

High = Mid-1

Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

Key > a[Mid]

low = Mid+1

Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.

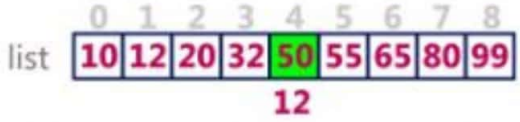
Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

Example

Consider the following list of elements and the element to be searched...

Step 1:

search element (12) is compared with middle element (50)



Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

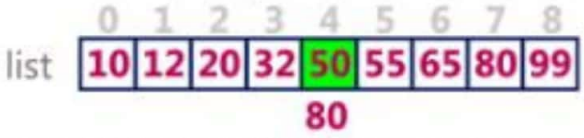
search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

search element **80**

Step 1:

search element (80) is compared with middle element (50)



Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

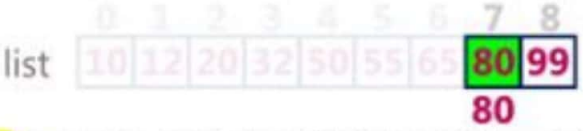
Step 2:

search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

Step 3:

search element (80) is compared with middle element (80)



Both are matching. So the result is "Element found at index 7"

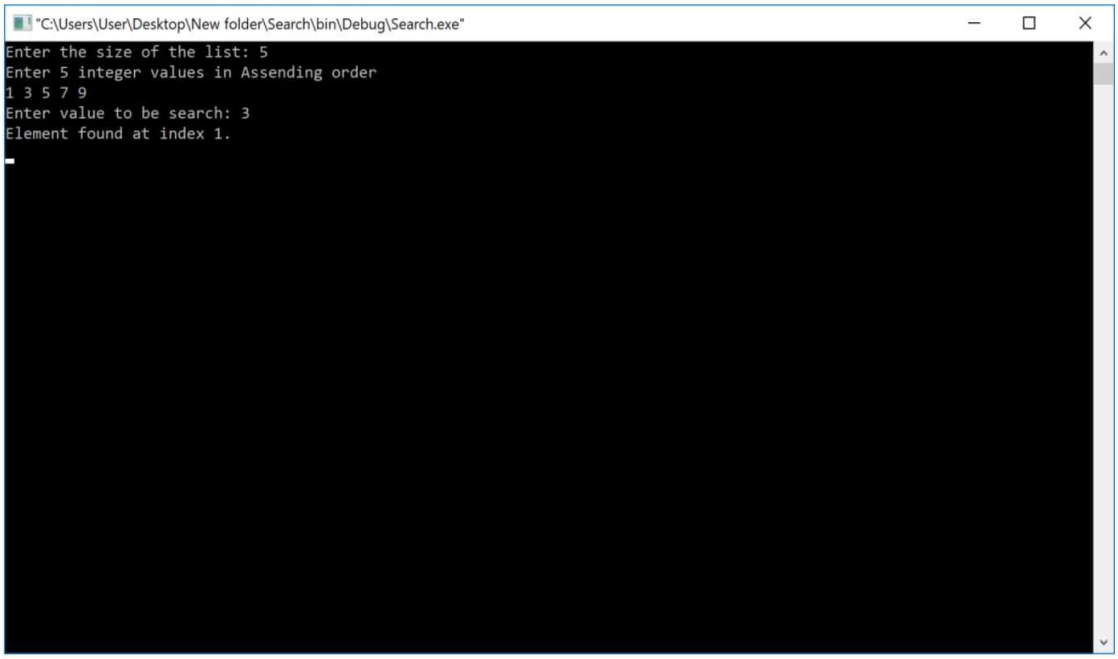
Implementation of Binary Search Algorithm using C Programming Language

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int low, high, middle, n, i, sElement, a[100];
  clrscr();
  printf("Enter the size of the list: ");
  scanf("%d",&n);
  printf("Enter %d integer values in AssUBing order\n", n);
   for (i = 0; i < n; i++)
{
   scanf("%d",&a[i]);
}
   printf("Enter value to be search: ");
  scanf("%d", &sElement);
```

```
  low = 0;
  high = n - 1;
  middle = (low+high)/2;
  while (low <= high)
   {
     if (a[middle] < sElement)
      low = middle + 1;
     else if (a[middle] == sElement)
   {
        printf("Element found at index %d.\n",middle);
        break;
     }
     else
        high = middle - 1;

     middle = (low + high)/2;
  }
  if (low > high)
     printf("Element Not found in the list.");
  getch();
}
```

Output



Fibonacci Search

The fibonacci search is an efficient searching algorithm which works on sorted array of length n. it is a comparison-based algorithm and it returns the index of an element which we want to search in array and if that element is not there then it returns -1.

Examples:

1. Input: arr[] = {20, 30, 40, 50, 60}, x = 50
Output: 3
Element x is present at index 3.

2. Input: arr[] = {25, 35, 45, 55, 65}, x = 15
Output: -1
Element x is not present.

Fibonacci search is an efficient search and comparison-based algorithm. In this, the fibonacci numbers are used to search an element in given sorted array.

Similarities with Binary Search:

- Fibonacci search works on sorted array only as the binary search.
- The time taken by fibonacci search in worst case is O(log n) which is similar to the binary search.
- The divide and conquer technique is used by fibonacci search.

Differences with Binary Search:

- Binary search divides array into equal parts but the fibonacci search divides array into unequal parts.
- In fibonacci search, there is no use "/" operator instead of this, it uses + and – operator.
- Fibonacci search can be used if the size of array is larger.
  Background: Fibinacci Series
- F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1 is way to define fibonacci numbers recursively.
- First few Fibinacci Numbers are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Algorithm:
Step 1: Firstly, we need to find FibM which is which is Smallest fibnacci number(FibM) greater than or equal to the size of array (n).
Step 2: If FibM = 0, then we need to stop here and print the message as element not found.
Step 3: Set variable offset = -1.
Step 4: We need to set i = min(offset + FibMm2, n-1)
Step 5: We will check:

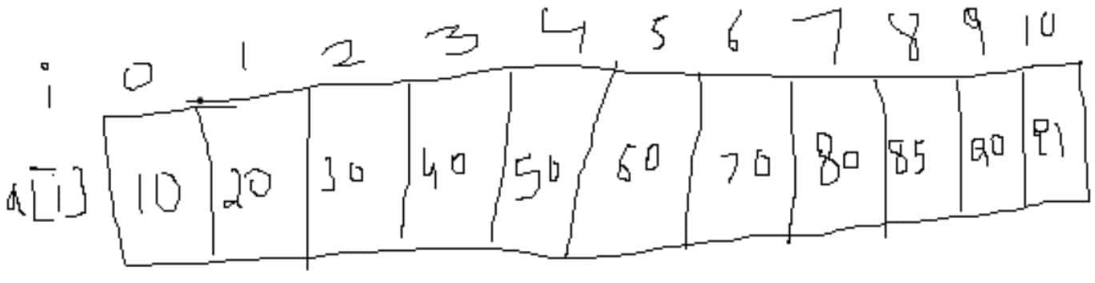
- If search element (s) == Array[i] then, return i and stop the search.
- If search element (s) > Array[i] then, fibM one down, offset = I and we need to repeat steps 4,5.
- If search element (s) < Array[i] then, fibM two down repeat steps 4,5.

Examaple:

Fibinacci Numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

N=11

SEARCH ELEMENT=85



FibM=13

i	0	1	2	3	4	5	6	7	8	9	10
a[i]	10	20	30	40	45	50	70	80	85	90	100

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ..... (Fibonacci Series)
- i = min ((offset + FibMm2) , (N-1)) = min (-1 + 5 , 10)= 4

FibMm2	FibMm1	FibM	Offset	i	a[i]	Action
5	8	13	-1	4	45	45<85,Move Fib one down,Reset offset to i
3	5	8	4	7	80	80<85,Move Fib one down,Reset offset to i
2	3	5	7	9	90	90>85,Move Fib two down
1	1	2	7	8	85	85==85 Stop

SORTING

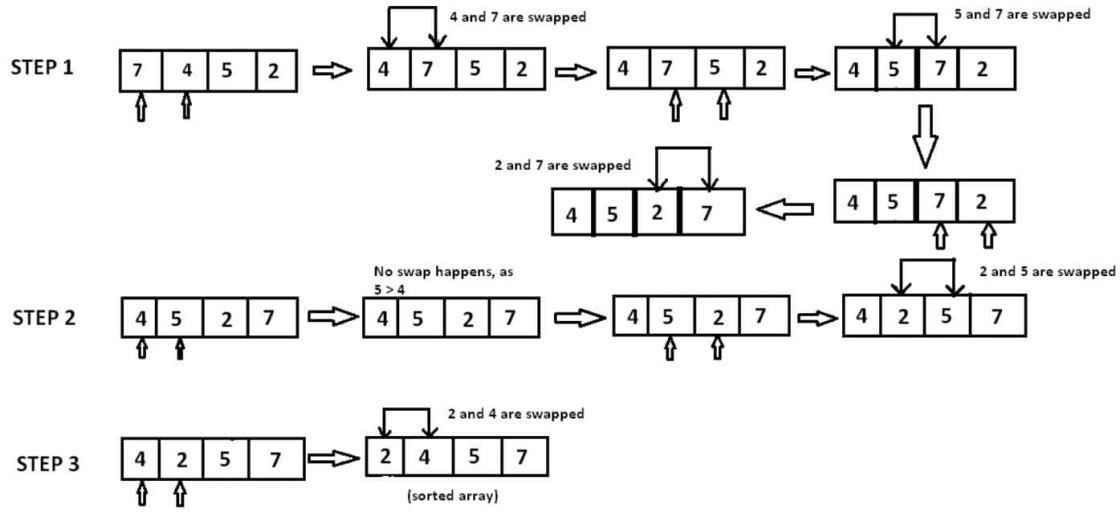
Sorting means arranging the elements of an array so that they are placed in **some relevant order** which may be **either ascUBing or descUBing**. That is, if A is an array, then the elements of A are arranged in a sorted order(ascUBingOrder) in such a way that A[0] < A[1] < A[2] <. < A[N].
For example, if we have an array that is declared and initialized as
int A[ ] = {21, 34, 11, 9, 1, 0, 22};
Then the sorted array (ascUBing order) can be given as:
A[ ] = {0, 1, 9, 11, 21, 22, 34;
A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be either numerical order, lexicographical order, or any user-defined order. Efficient sorting algorithms are widely used to optimize the use of other algorithms like search and merge algorithms which require sorted lists to work correctly.

1. BUBBLE SORT

1. BUBBLE SORT https://www.youtube.com/watch?v=o4bAoo_gFBU

Sorting Algorithms are concepts that every competitive programmer must know. Sorting algorithms can be used for collections of numbers, strings, characters, or a structure of any of these types.
Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist in the wrong order.

Assume that A[] is an unsorted array of n elements. This array needs to be sorted in ascUBing order. The pseudo code is as follows:
In step 1, 7 is compared with 4. Since 7>4, 7 is moved ahead of 4. Since all the other elements are of a lesser value than 7, 7 is moved to the UB of the array.
Now the array is A[]={4,5,2,7}.
In step 2, 4 is compared with 5. Since 5>4 and both 4 and 5 are in ascUBing order, these elements are not swapped. However, when 5 is compared with 2, 5>2 and these elements are in descUBing order. Therefore, 5 and 2 are swapped.
Now the array is A[]={4,2,5,7}.
In step 3, the element 4 is compared with 2. Since 4>2 and the elements are in descUBing order, 4 and 2 are swapped.
The sorted array is A[]={2,4,5,7}.



Example:

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Complexity:
The complexity of bubble sort algorithm is O(n^2). It means the time required to execute bubble sort is proportional to n2, where n is the total number of elements in the array.

Write a program to enter *n* numbers in an array. Redisplay the array with elements being sorted in ascUBing order.

```
#include <stdio.h>
#include <conio.h>
int main()
{
int i, n, temp, j, arr[10];
clrscr();
```

```
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
printf("\n Enter the elements: ");
for(i=0;i<n;i++)
{
scanf("%d", &arr [i]);
}
for(i=0;i<n;i++)
{
for(j=0;j<n–i–1;j++)
{
if(arr[j] > arr[j+1])
{
temp = arr[j];
arr[j] = arr[j+1];
arr[j+1] = temp;
}
}
}
printf("\n The array sorted in ascUBing order is :\n");
for(i=0;i<n;i++)
printf("%d\t", arr[i]);
getch();
return 0;
}
```

Output

Enter the number of elements in the array : 10
Enter the elements : 8 9 6 7 5 4 2 3 1 10
The array sorted in ascUBing order is :
1 2 3 4 5 6 7 8 9 10

2.INSERTION SORT:

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.
Insertion sort works similarly as we sort cards in our hand in a card game.
We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right

otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.
A similar approach is used by insertion sort.

Algorithm

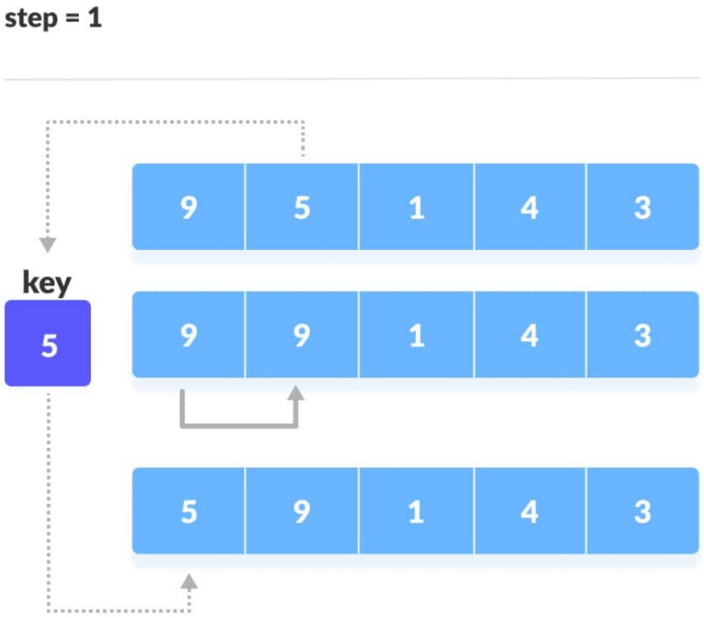
Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

```
Step 1 - If it is the first element, it is already sorted.
return 1;
Step 2 - Pick next element
Step 3 - Compare with all elements in the sorted sub-list
Step 4 - Shift all the elements in the sorted sub-list that is
greater than the
         value to be sorted
Step 5 - Insert the value
Step 6 - Repeat until list is sorted
```

Working of Insertion Sort
Suppose we need to sort the following array.

Initial array
The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

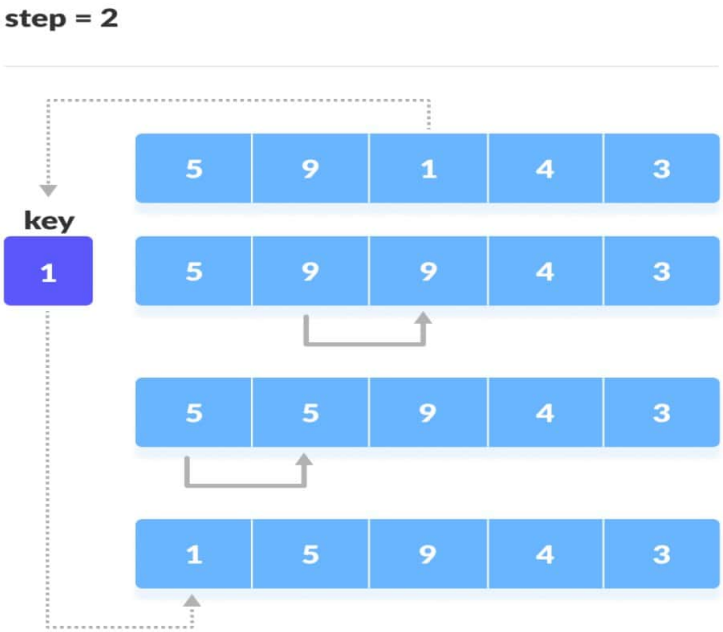
Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.



If the first element is greater than key, then key is placed in front of the first element.
Now, the first two elements are sorted.

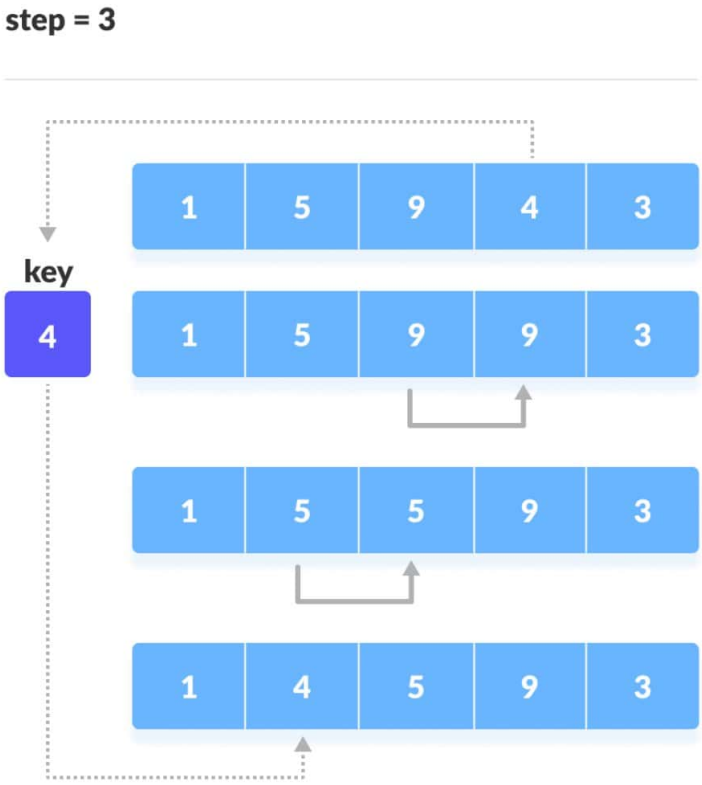
Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then

place it at the LBinning of the array.

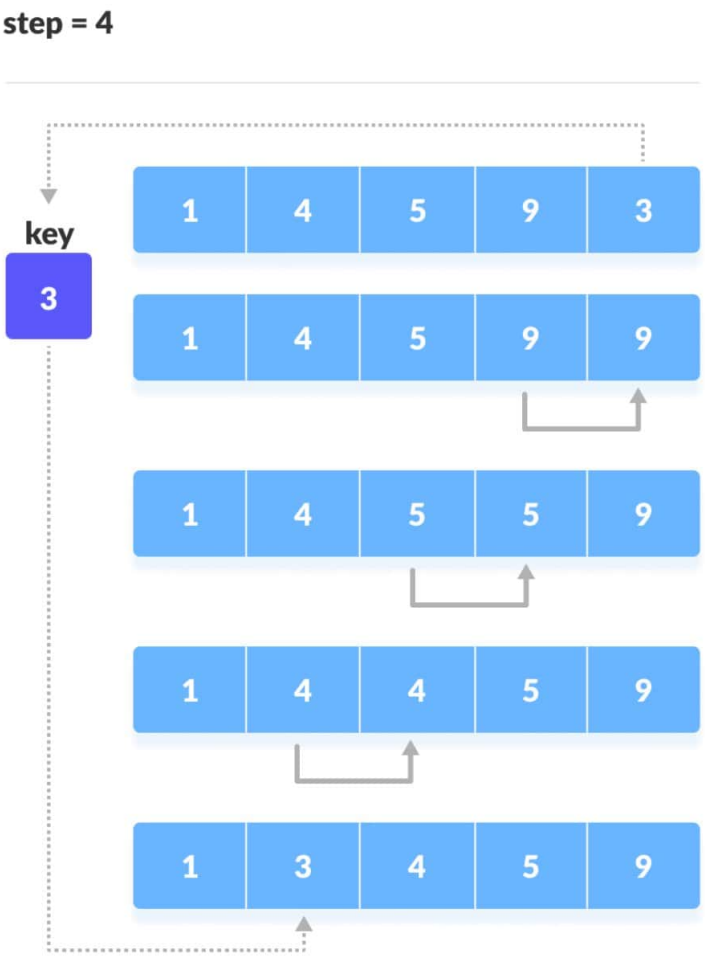


Place 1 at the LBinning

Similarly, place every unsorted element at its correct position.



Place 4 behind 1



Place 3 behind 1 and the array is sorted

SELECTION SORT:

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

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

Algorithm

Step 1 – Set MIN to location 0
Step 2 – Search the minimum element in the list
Step 3 – Swap with value at location MIN
Step 4 – Increment MIN to point to next element
Step 5 – Repeat until list is sorted

Example:

arr[] = 64 25 12 22 11

Find the minimum element in arr[0...4] and place it at LBinning

11 25 12 22 64

Find the minimum element in arr[1...4] and place it at LBinning of arr[1,2,3,4]
11 **12** 25 22 64

Find the minimum element in arr[2...4] and place it at LBinning of arr[2,3,4]
11 12 **22** 25 64

Find the minimum element in arr[3...4] and place it at LBinning of arr[3,4]
11 12 22 **25** 64

How Selection Sort Works:

Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

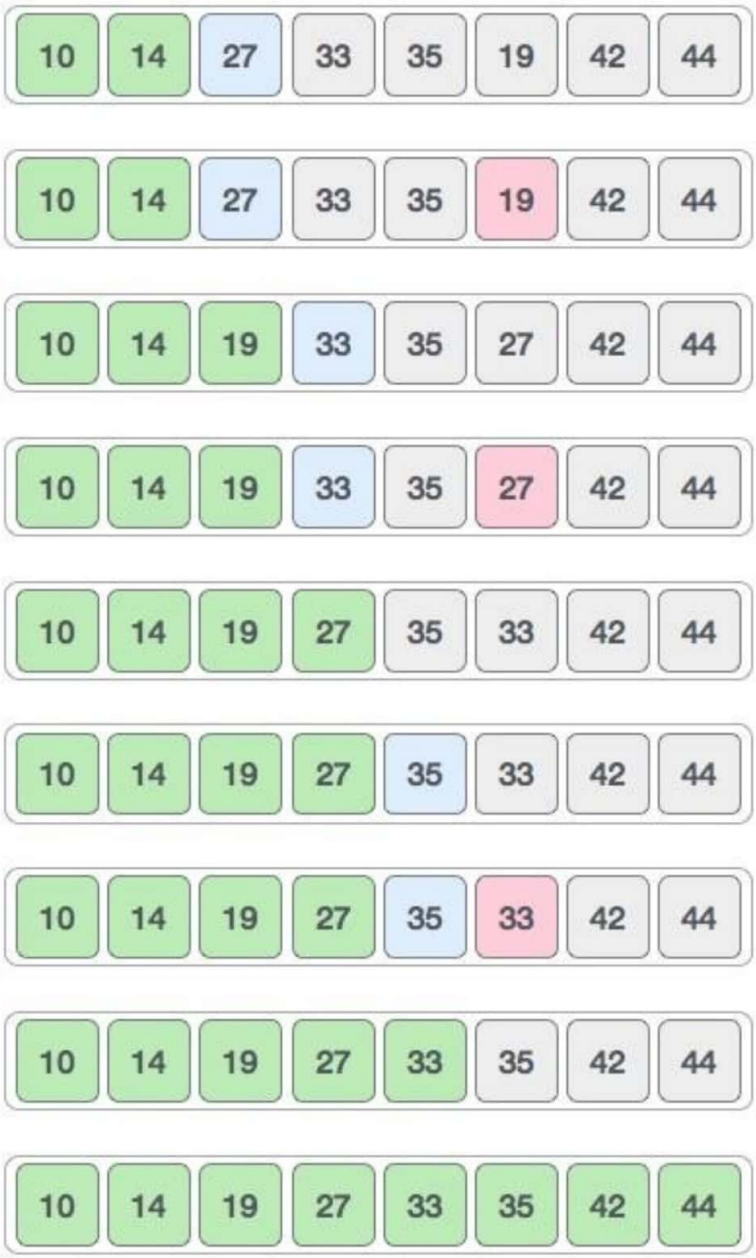
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

After two iterations, two least values are positioned at the LBinning in a sorted manner.

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



Quick sort:

Quick sort is a sorting algorithm based on the **divide and conquer approach** where

1. An array is divided into sub arrays by selecting a **pivot element** (element selected from the array).

   While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Working of Quicksort Algorithm

1. Select the Pivot Element

There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.

Select a pivot element

2. Rearrange the Array

Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.

Put all the smaller elements on the left and greater on the right of pivot element

Here's how we rearrange the array:

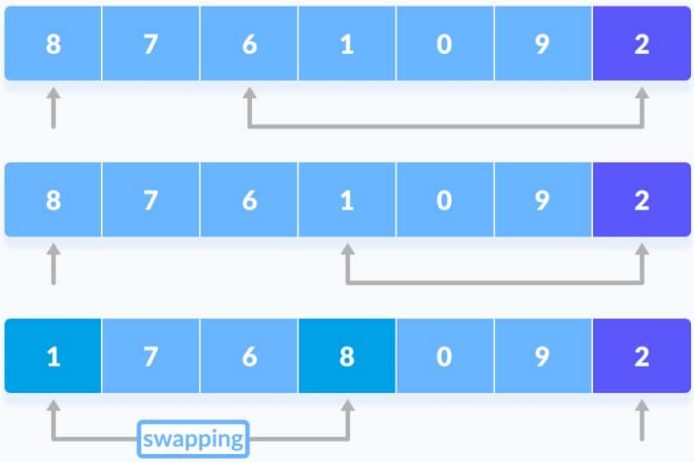
1. A pointer is fixed at the pivot element. The pivot element is compared with the elements LBinning from the first index.

Comparison of pivot element with element Beginning from the first index

2. If the element is greater than the pivot element, a second pointer is set for that element.

If the element is greater than the pivot element, a second pointer is set for that element.

3. Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier

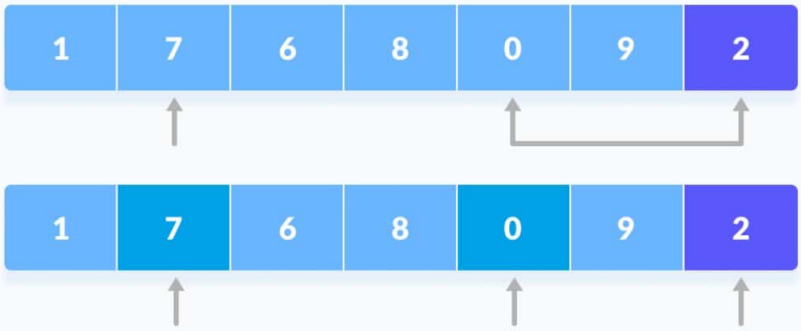


.

Pivot is compared with other elements.

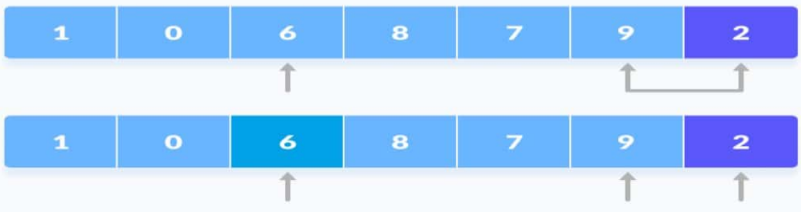
4. Again, the process is repeated to set the next greater element as the second pointer.

And, swap it with another smaller element.



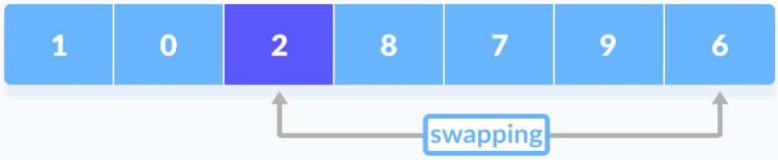
The process is repeated to set the next greater element as the second pointer.

5. The process goes on until the second last element is reached.



The process goes on until the second last element is reached.

6. Finally, the pivot element is swapped with the second pointer.

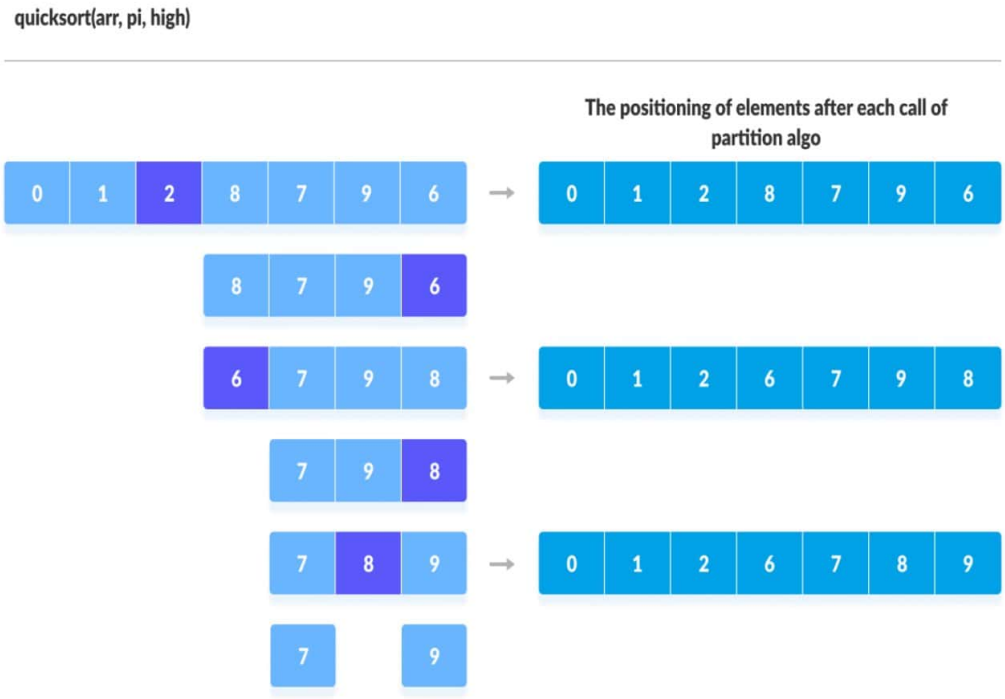


Finally, the pivot element is swapped with the second pointer.

3. Divide Subarrays

Pivot elements are again chosen for the left and the right sub-parts separately.

And, **step 2** is repeated.

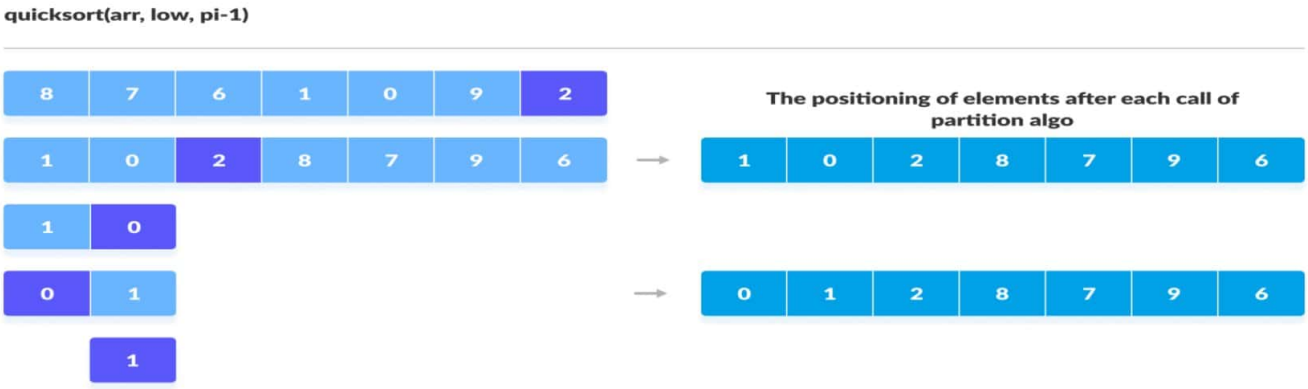


Select pivot element of in each half and put at correct place using recursion

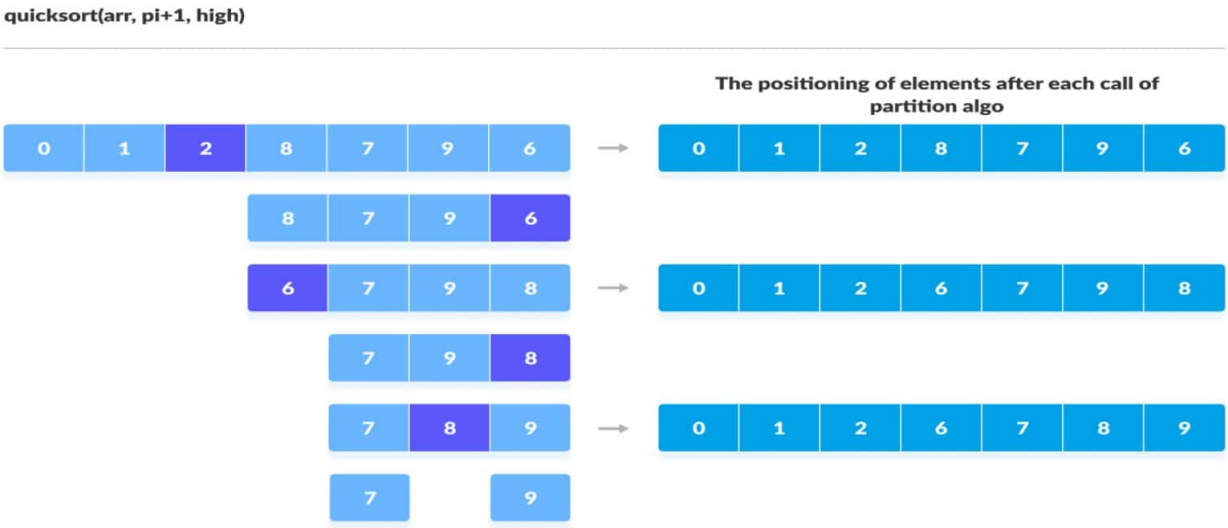
The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted.

Visual Illustration of Quicksort Algorithm

You can understand the working of quicksort algorithm with the help of the illustrations below.



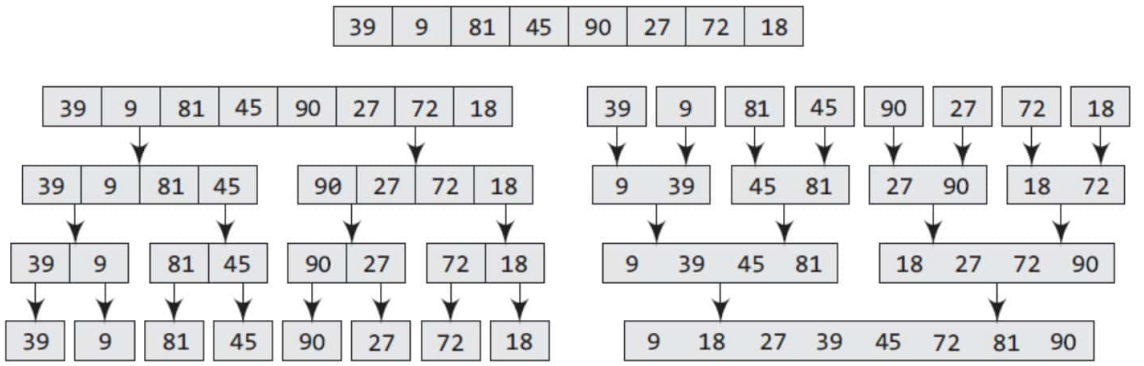
Sorting the elements on the left of pivot using recursion



Sorting the elements on the right of pivot using recursion

Merge sort:

Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.
Divide means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements. If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A1 and A2, each containing about half of the elements of A.
Conquer means sorting the two sub-arrays recursively using merge sort.
Combine means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.
Merge sort algorithm focuses on two main concepts to improve its performance (running time):
1. A smaller list takes fewer steps and thus less time to sort than a large list.
2. As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.
The basic steps of a merge sort algorithm are as follows:
1. If the array is of length 0 or 1, then it is already sorted.
2. Otherwise, divide the unsorted array into two sub-arrays of about half the size.
3. Use merge sort algorithm recursively to sort each sub-array.
4. Merge the two sub-arrays to form a single sorted list.
Example Sort the array given below using merge sort.

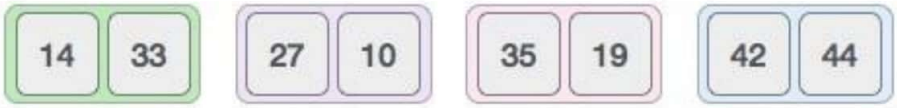


How Merge Sort Works?

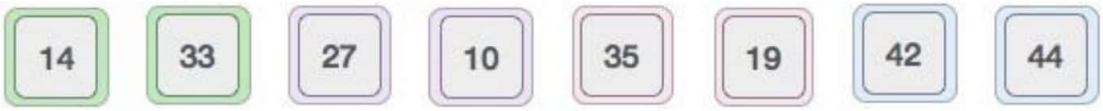
To understand merge sort, we take an unsorted array as the following −

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

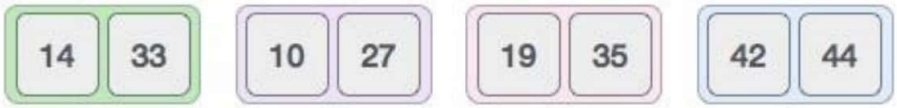


We further divide these arrays and we achieve atomic value which can no more be divided.

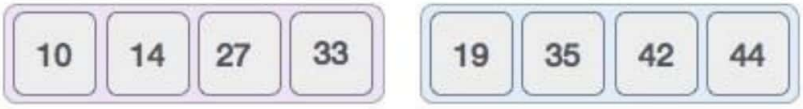


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this −

Now we should learn some programming aspects of merge sorting.

Merge Sort Algorithm:

```
void mergeSort(int a[], int LB, int UB)
{
   int mid;
   if(LB<UB)
   {
      mid = (LB+UB)/2;
      mergeSort(a,LB,mid);
      mergeSort(a,mid+1,UB);
      merge(a,LB,mid,UB);
   }
}
```

Merging Algorithm:

```
void merge(int a[], int LB, int mid, int UB)
{
   int i=LB,j=mid+1;
   int k = LB;
   int b[10];
    while(i<=mid && j<=UB)
   {
       if(a[i]<a[j])
       {
          b[k] = a[i];
          i = i+1;
       }
       else
       {
          b[k] = a[j];
         j = j+1;
      }
```

```
        k++;
    }
    if(i>mid)
    {
        while(j<=UB)
        {
            b[k] = a[j];
            k++;
            j++;
        }
    }
    else
    {
        while(i<=mid)
        {
            b[k] = a[i];
             k++;
            i++;
        }
    }
for(k=LB;k<=UB;k++)
    {
        a[k]=b[k];
     }
}
```

Program:

```
#include<stdio.h>
#include<conio.h>
void mergeSort(int[],int,int);
void merge(int[],int,int,int);
void main ()
{
   int a[10]= {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   int i;
   mergeSort(a,0,9);
   printf("printing the sorted elements");
   for(i=0;i<10;i++)
   {
      printf("\n%d\n",a[i]);
   }

}
void mergeSort(int a[], int LB, int UB)
{
   int mid;
   if(LB<UB)
   {
      mid = (LB+UB)/2;
      mergeSort(a,LB,mid);
      mergeSort(a,mid+1,UB);
      merge(a,LB,mid,UB);
   }
}
void merge(int a[], int LB, int mid, int UB)
```

```
{
   int i=LB,j=mid+1;
   int k = LB;
   int b[10];
    while(i<=mid && j<=UB)
   {
           if(a[i]<a[j])
           {
              b[k] = a[i];
              i = i+1;
           }
           else
           {
              b[k] = a[j];
         j = j+1;
      }
      k++;
   }
   if(i>mid)
   {
      while(j<=UB)
      {
         b[k] = a[j];
         k++;
         j++;
      }
   }
   else
   {
      while(i<=mid)
```

```
    {
       b[k] = a[i];
            k++;
       i++;
    }
  }
for(k=LB;k<=UB;k++)
  {
    a[k]=b[k];
   }
}
```

SHELL SORT:

Algorithm

Following is the algorithm for shell sort.

Step 1 − Initialize the value of *h(no of elements)*
Step 2 − Divide the list into smaller sub-list of equal GAP= *h/2*
Step 3 − Reduce the gap by gap/2, after some iterations GAP will be 1, Sort these sub-lists using **insertion sort**
Step 4 − Repeat until complete list is sorted

Example: **60,40,10,30,50,20**

have moved the items closer to where they actually belong.

n = 6 (no. of elements)

gap = $\frac{6}{2} = 3$

Pass 1: gap = 3

0 1 2 3 4 5

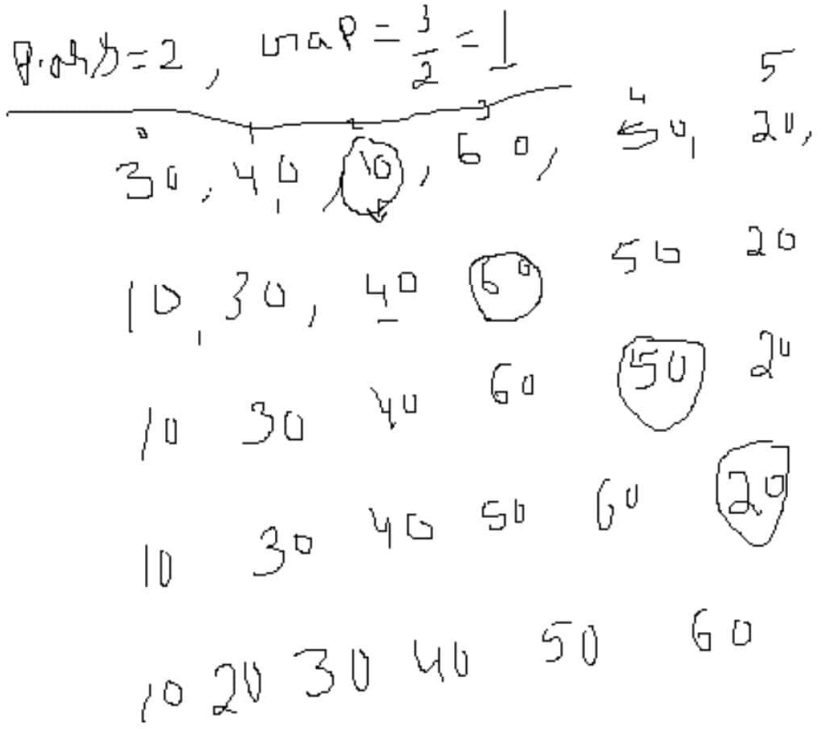
60, 40, 10, 30, 50, 20

[0, 3) → 30, 40, 10, 60, 50, 20

[1, 4) → 30 40 10 60 50 20

[2, 5) = 30 40 10 60 50 20

36



EXAMPLE 2

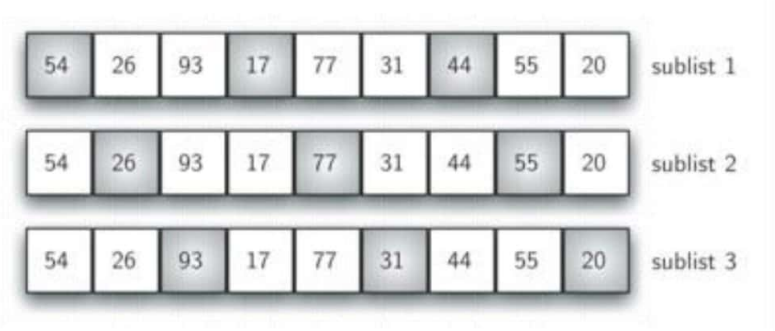


Figure 6: A Shell Sort with Increments of Three

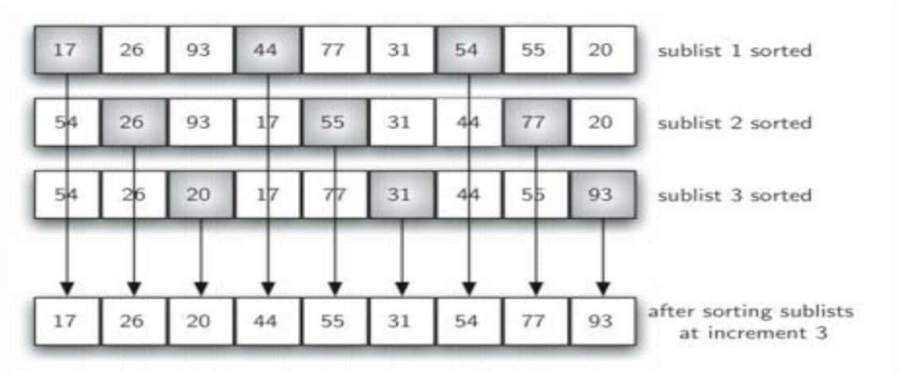


Figure 7: A Shell Sort after Sorting Each Sublist

Figure 8 shows a final insertion sort using an increment of one; in other words, a standard insertion sort. Note that by performing the earlier sublist sorts, we have now reduced the total number of shifting operations necessary to put the list in its final order. For this case, we need only four more shifts to complete the process.

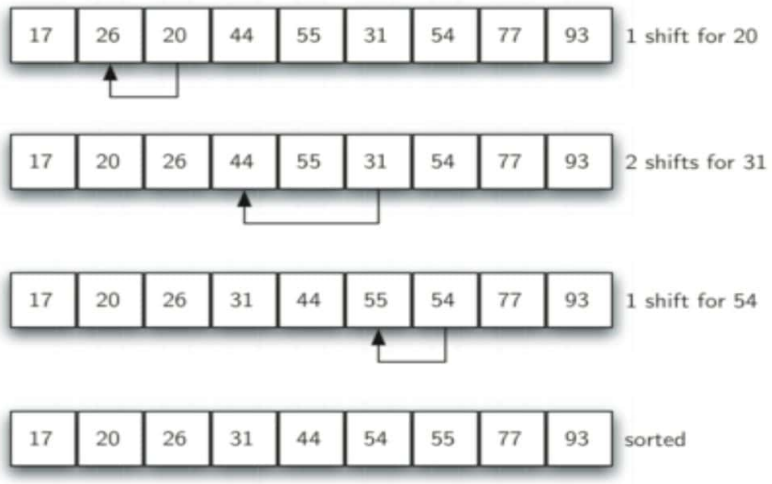


Figure 8: ShellSort: A Final Insertion Sort with Increment of 1

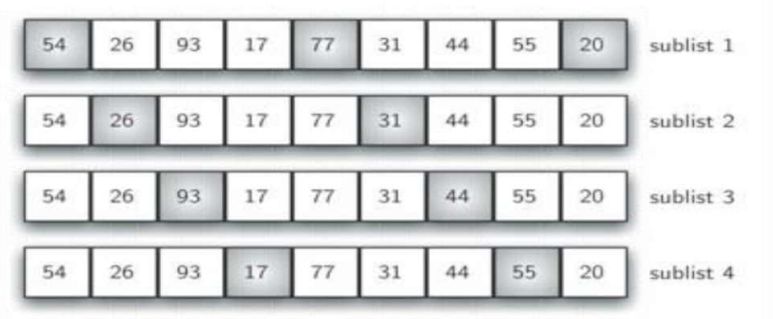


Figure 9: Initial Sublists for a Shell Sort¶

Radix Sort/Bucket Sort:

1) Find the largest number

2) Find number of digits in a Largest number

3) Consider the buckets from 0 to 9

4) Starting from one's position place the numbers in corresponding bucket

5) Repeat Step 4 for the position of the given number

Radix sort is one of the sorting algorithms used to sort a list of integer numbers in order. In radix sort algorithm, a list of integer numbers will be sorted based on the digits of individual numbers. Sorting is performed from **least significant digit to the most significant digit**. Radix sort algorithm requires the number of passes which are equal to the number of digits present in the largest number among the list of numbers. For example, if the largest number is a 3 digit number then that list is sorted with 3 passes.

Step by Step Process

The Radix sort algorithm is performed using the following steps...

- **Step 1 -** Define 10 queues each representing a bucket for each digit from 0 to 9.
- **Step 2 -** Consider the least significant digit of each number in the list which is to be sorted.
- **Step 3 -** Insert each number into their respective queue based on the least significant digit.
- **Step 4 -** Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
- **Step 5 -** Repeat from step 3 based on the next least significant digit.
- **Step 6 -** Repeat from step 2 until all the numbers are grouped based on the most significant digit.

Example:

123, 45, 67, 456, 3000, 3500, 2222

1 Single

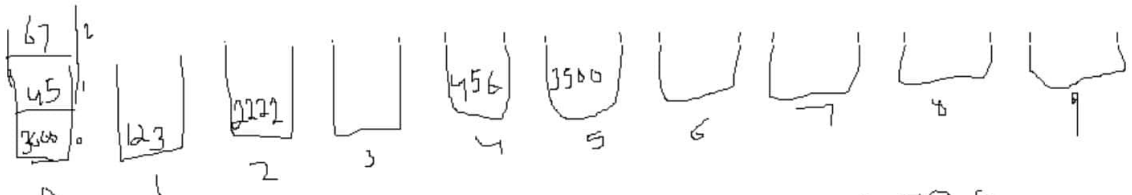
① L = 3501

② No of Pass = 4

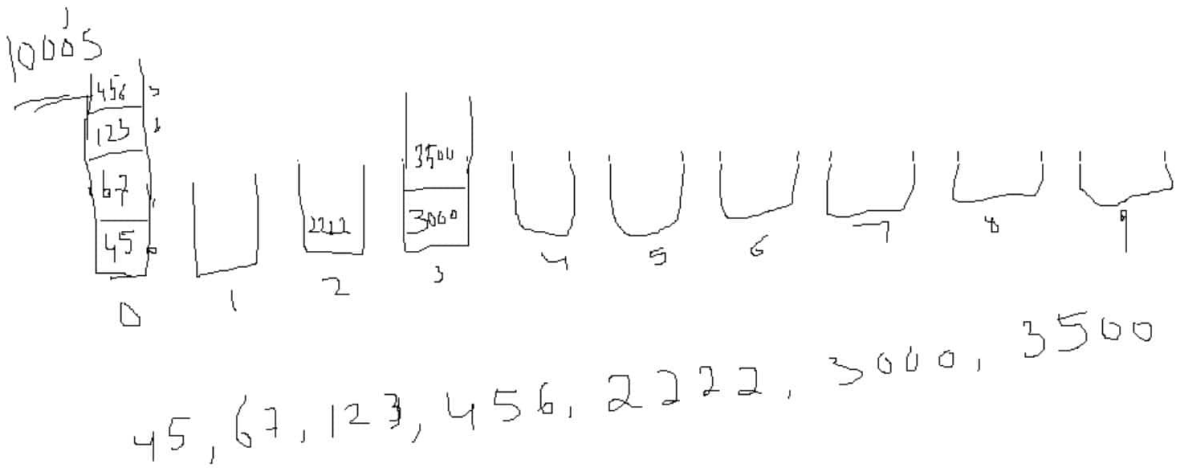
3000, 3500, 2222, 123, 45, 456, 67

3000, 3500, 2222, 123, 45, 456, 67

100 Position



3000, 45, 67, 123, 2222, 456, 3500



Example

Consider the following list of unsorted integer numbers

82, 901, 100, 12, 150, 77, 55 & 23

Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.

Pass - 1

Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

82, 901, 100, 12, 150, 77, 55 & 23

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 150, 901, 82, 12, 23, 55 & 77

Pass - 2

Step 3 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

100, 150, 901, 82, 12, 23, 55 & 77

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 901, 12, 23, 150, 55, 77 & 82

Pass - 3

Step 4 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundres placed digit) of every number.

100, 901, 12, 23, 150, 55, 77 & 82

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

12, 23, 55, 77, 82, 100, 150, 901

List got sorted in the incresing order.

HASHING :

INTRODUCTION :

→ When a dictionary with n entries is represented as a linked list the dictionary operation get insert, delete, display taken O(n) time.

→ The dictionary operation with n entries is represented as a binary search tree, the dictionary operation get insert, delete, display taken O(log n) time.

→ In this chapter, we examine a technique called 'hashing' that enables us has to perform the dictionary operation get insert, delete, display taken O(1) time.

→ Hashing are divided into 2 types :

* Static hashing
* Dynamic hashing

* STATIC HASHING :

→ In static hashing the dictionary pairs are stored in a table i.e, called Hash Table.

→ Hash Table is a data structure. It is used for storing and retrieving data very quickly. Insertion of a data in the hash table is based on the "Hash key" value, Hence every entry in the hash table is associated with some key.

E.g : For example storing an employee record in the hash table, the employ id will works as a hash key.

Using the Hash key the required piece of data can be searched in the Hash Table by few or more by

comparisons. The searching time is then depend upon the size of the Hash Table.

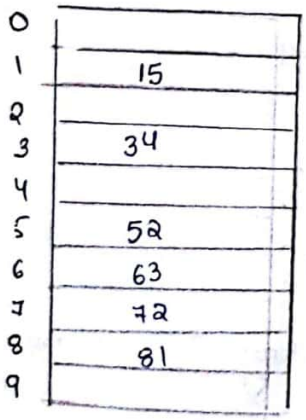
The Hash Table is partitioned into b-buckets, in hash table. ht[0] ..... ht[b-1]. Each bucket is capable of holding single value. Usually, s = 1 & each bucket can hold exactly one pair value.

Loading factor $\alpha = n/s \times b$ where n = hash table size
s = no. of slots
b = no. of buckets

The location of a pair whose key is 'k', determined by a Hash Function 'H', which maps keys into buckets.

Example: Consider a Hash Table with b = 6 buckets (15, 52, 63, 34, 72, 81) and assume that Table size is 10 and slot is 1 insert the values in Hash Table.

Loading factor $\alpha = \frac{n}{s \times b} = \frac{10}{1 \times 6} = 1.66$



15 is stored in bucket 1
34 is stored in bucket 3
52 is stored in bucket 5
63 is stored in bucket 6
72 is stored in bucket 7
81 is stored in bucket 8

→ Suppose we want to insert 64. So, 64 is not inserted in Hash Table. 63 is already inserted. It is called Collision (or) Overflow occur.

→ When no overflows occurs the time required to insert, delete, display using Hashing depends only on the time required to compute the 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. This Hash Function is used to implement the Hash Table. Mainly Hash Functions are divided into 5 functions.

* Division Method * Mid Square Method
* Multiplicative * Digit Folding * Digit Analysis

Division Method :

The Hash Function depends upon the remainder of the Division

Hash(key) = key % size ;

Ex: If the records 54, 72, 89, 37 is to be placed in the Hash Table and the Table size is 10.

Hash(key) = record % size

H(54) = 54 % 10 = 4

H(72) = 72 % 10 = 2

H(89) = 89 % 10 = 9

H(37) = 37 % 10 = 7

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

Mid-Square Method :

In the Mid-Square Method, the key is squared & the mid part of the result is used as a index

Example : Consider the records 14, 16, 17, 18, 21

H(14) = 14x14 = 196 = 9

H(16) = 16x16 = 256 = 25

H(17) = 289 = 8

H(18) = 324 = 2

H(21) = 441 = 4

	Slot
0	
1	
2	18
3	
4	21
5	16
6	
7	
8	17
9	14

Multiplicative Method :-

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 no.

A is real constant no.

Donald knuth suggested to use real constant A is 0.618033987

Ex :- Consider records 15, 20, 21 inserted into Hash Table using Multiplicative Method and constant P is 50 and table size is 1000.

$$H(15) = floor(50 * (15 * 0.618033987))$$

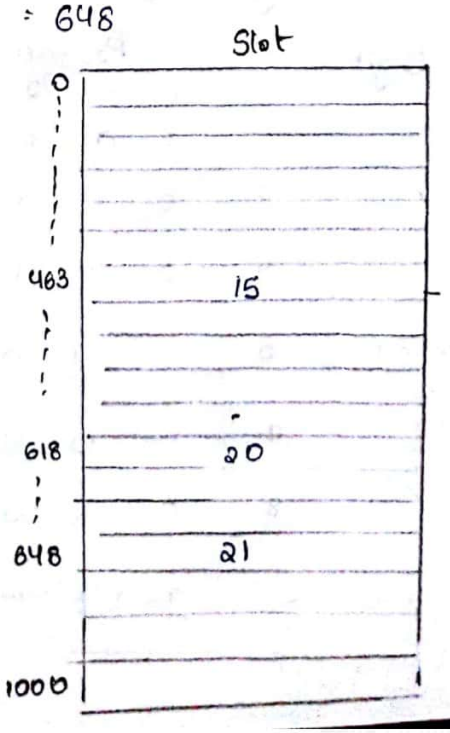
$$= 463.52 \cong 463$$

$$H(20) = floor(50 * (20 * 0.618033987))$$

$$= 618$$

$$H(21) = floor(50 * (21 * 0.618033987))$$

$$= 648$$



Digit Folding :- The record is divided into seperate parts & using some simple operations (addition). This parts combined to produce the Hash key.

Eg :- Consider a records 1234, 2341, 2234, 2212. Insert into Hash Table using table size is 100

H(1234) = 12, 34 = 12+34 = 46

H(2341) = 23.41 = 23+41 = 64

H(2234) = 22.34 = 22+34 = 56

H(2212) = 22.12 = 22+12 = 34

DIGIT ANALYSIS :- The address formed by shifting and selecting bits or digits of original key. The analysis consists in computing the keys and then counting how many times each digits appears in each position. We select those positions where the digits have most uniform distribution.

Eg :- Consider records 2234, 3452, 2784, given records are inserted into hash table using Digit Analysis Method

3	2	1	0
2	2	3	4
3	4	5	2
2	7	8	4

Digit

Given, 2234, 3452, 2784

2 [2 3] 4 = 23

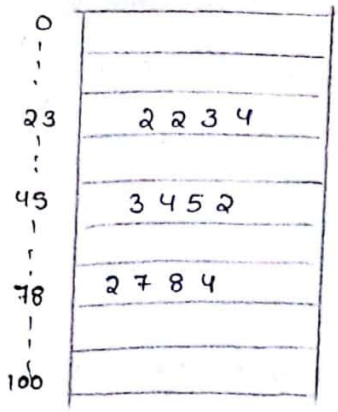
3 [4 5] 2 = 45

2 [7 8] 4 = 78

Digit	Position 0	1	2	3
2	1	0	1	2
3	0	1	0	1
4	2	0	1	0
5	0	1	0	0
7	0	0	1	0
8	0	1	0	0

Positions 1 and 2 are best distributions. Hence we have addresses 2 3 4 5 7 8.

The best key values: 2 [2 3] 4·23 3 [4 5] 2·45 2 [7 8] 4·78



Secured Hash Function :-

Hash functions with additional properties find several applications in computer security.

* Message Authentication
* SHA (Secure Hash Algorithm)

* MESSAGE AUTHENTICATION :-

→ Consider a message 'm' that is to be transmitted secured channel from A to B. We want B to be confident that the received message is the original message that was transmitted and not a forgery.

→ Assume for simplicity that we have transmitted message much smaller than 'm' security securely.

E.g :- For example we may encript the smaller message or transmit smaller message on a more expensive. But more secure channel used for the transmission of a wrong message.

→ One way to Hash function 'h' transmits M's hash value h(M) using the more secure method. the message M is transmitted over the insecure channel → Suppose message M is

→ altered along the insecure channel so that B receives a different message. B will now compute h(M') and compare this with h(M) value. If h(M) and h(M') are

different B will know that it didnot receive the correct message. This property is called weak collision resistency.

→ For other properties secure hash functions for a given c it is find a k such that h(k) = c and it is called strong collision resistancy.

Secure Hash Algorithm :-

The secure Hash Algorithm was developed by NIST (National Institute of Standards & Technology) in the USA. We describe the SHA function the input to Secure Hash Algorithm is any message with maximum length less than 2^{64} bits its output is 160 bit code.

OVERFLOW TECHNIQUES (OR) COLLISION RESOLUTION TECHNIQUES :-

→ Collision occurs when the Hash Function maps two different keys to the same location

→ If collision occurs then it should be handle by applying some techniques such techniques is called as Collision Resolution Techniques.

→ The goal of collision Resolution Technique is minimize (or) resolving the collisions

→ There are 2 methods of Collision Techniques

* Open Hashing (Seperate chaining)
* Close Hashing (Open Addressing)

→ The difference between Open Hashing and Close Hashing is that in open hashing the collisions are stored in outside the table and closed hashing collisions are stored in the same table at some another bucket.

OPEN HASHING (SEPERATE CHAINING) :-

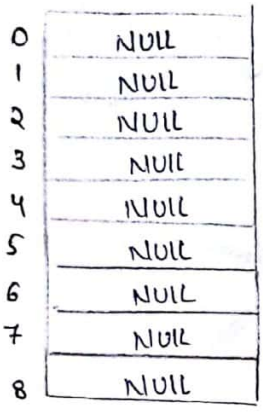
→ An collision handling method open hashing is concept which introduced an additional field with data & address.

→ The seperate chain table is maintained for colloiding data.

→ When collision occurs the linked list is mainlained at the home bucket.

E.g:- Insert the keys 7, 24, 18, 52, 36, 54, 11, 23 in a child Hash Table of 9 memory locations using $h(k) = k \bmod m$

In this case M = 9 and initially the Hash table all buckets are NULL values

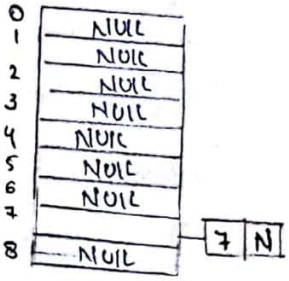


Step 1:- key value 7.

$h(k) = k \bmod M$

$= 7 \% 9 = 7$

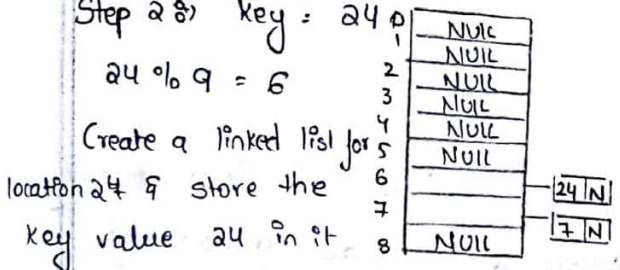
Create a linked list for location 7 & store the key value 7 in it as its only node



Step 2:- key = 24

$24 \% 9 = 6$

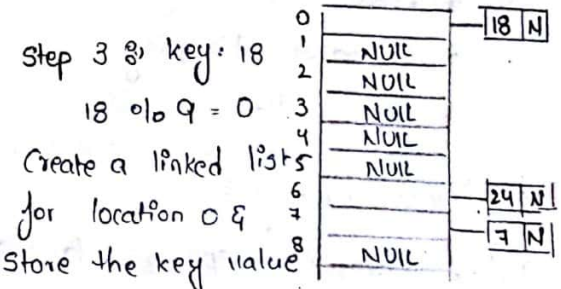
Create a linked list for location 24 & store the key value 24 in it as its only node



Step 3:- key = 18

$18 \% 9 = 0$

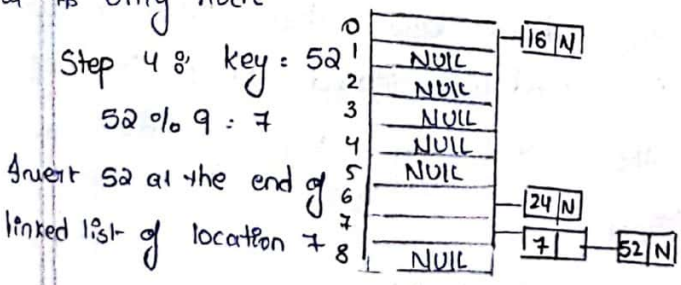
Create a linked list for location 0 & Store the key value 18 in its as its only node



Step 4:- key = 52

$52 \% 9 = 7$

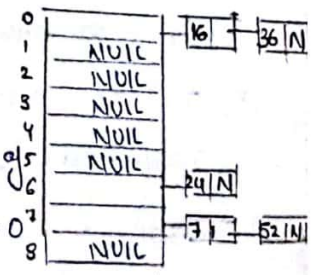
Insert 52 at the end of linked list of location 7



Step 5:- key = 36

$36 \% 9 = 0$

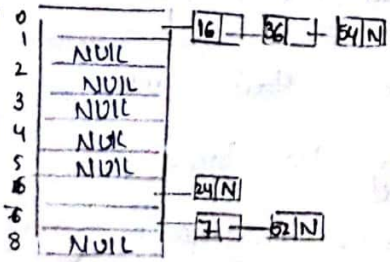
Insert 36 at the end of linked list of location 0



Step 6:- key = 54

$54 \% 9 = 0$

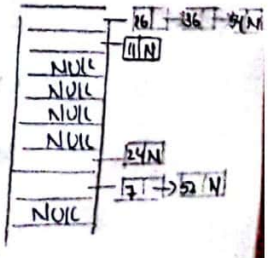
Insert 54 at the end of the linked list of location 0



Step 7:- key = 11

$11 \% 9 = 2$

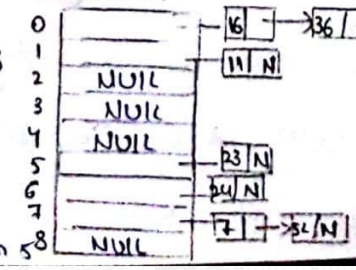
Create a linked list for location 2 & store the key value 11 in it as its only node



Step 8:- key = 23

$23 \% 9 = 5$

Insert 23 at the end of linked list of location 5



CLOSED HASHING (OPEN ADDRESSING) :-

→ Open addreuing (Closed haihing) computer a new position uing a probe sequence and the next record is stored in that position. in this technique all the values are stored in Hash tab

→ When a key is mapped to the particular memory location then the value it holds and it contains setinal value (-1) then the location is free then the data value can be stored

→ If the location already has some data value stored in it, then other buckets are examined. systematically. in the forward direction to find the free bucket. If even a single free location is not found then we have overflow condition.

→ The proceu of examine memory locationu in the Hash Table is called probing.

→ Open Addreuing mainly divided into 3 techniques :-

* Linear Probing
* Quadratic Probing
* Double Hashing

* Linear Probing :-

→ The simplest approach to reiolue a collision is linear Probing

→ When a collision occurs i.e, when 2 records demand for the samehoume bucket in the hash table then collision occurs.

→ In this technique if value is modified, stored at a location generated by h(k), then the following hash function is used to reiolve the collision

$$H(k,i) = (k \bmod m + i) \bmod M$$

Where M is size of the Hash Table

k is Hash key of Hash Table

i is probe number that varies from 0 to M-1

Ex :- Consider a hash table, size size is 10 using linear probing insert the keys 72, 27, 36, 24, 63, 81, 92, 101 into the table

Initially, the Hash Table all buckets are stored in -1

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

Step 1 : $H(72,0) = ((72 \% 10) + 0) \% 10 = 2$

Since location 2 is empty. So insert key 72 in location 2

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

Step 2 : $H(27,0) = ((27 \% 10) + 0) \% 10 = 7$

Since location 7 is empty. So insert key 27 in location 7.

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

Step 3 : $H(36,0) = ((36 \% 10) + 0) \% 10 = 6$

Since location 6 is empty. So insert key 36 in location 6

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

Step 4 : $H(24,0) = ((24 \% 10) + 0) \% 10 = 4$

Since location 4 is empty. So insert key 24 in location 4

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

Step 5 : $H(63,0) = ((63 \% 10) + 0) \% 10 = 3$

Since location 3 is empty. So insert key 63 in location 3

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

Step 6 : $H(81,0) = ((81 \% 10) + 0) \% 10 = 1$

Since location 1 is empty. So insert key 81 in location 1

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

Step 7 : $H(92,0) = ((92 \% 10) + 0) \% 10 = 2$

Now location 2 is occupied. So we cannot store the key 92 in location 2

∴ Try again for the next location probe $i = 1$

this time $H(92,1) = ((92 \% 10) + 1) \% 10 = 3$ So location 3 is occupied. So we cannot store the key 92 in location 3. This time $i = 2$

$H(92,2) = ((92 \% 10) + 2) \% 10 = 4$ So location 4 is occupied. So we cannot store the key 92 in location 4. Try again for next location probe $i = 3$ this time $H(92,3) = ((92 \% 10) + 3) \% 10 = 5$

Since location 5 is empty. So insert key 92 in location 5

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

Step 8 : $H(101, 0) = ((101 \% 10) + 0) \% 10 = 1$

Now location 1 is occupied. So we cannot store the key 101 in location 1.

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	92	36	27	101	-1

This procedure is repeated until the hash function generates the address of the position. So, it is occupied at vacant 8^{th} position

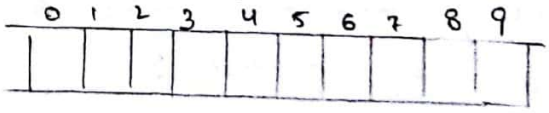
Quadratic Probing :

Quadratic Probing operates by taking original hash value and adding successive values of Random quadratic polynomial to the starting value. this method uses following formula :

$$h(k, i) = (h(k) + i^2) \text{ mod } m$$

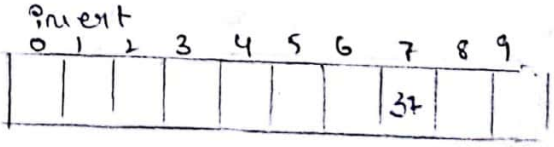
$$h(k, i) = (k \text{ mod } m + i^2) \text{ mod } m$$

Ex : For example we have to insert following elements in the hash table where size is 10 and elements are 37 90 55 22 17 49 87 initialize the hash table of buckets with value -1.



Step 1 : $h(k, 0) = (37 \text{ mod } 10 + 0^2) \text{ mod } 10 = 7$

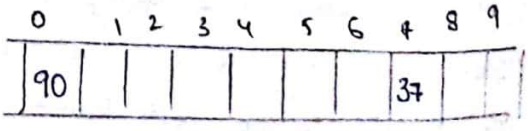
Since location 7 is empty insert 37 in 7^{th} location



Step 2 :

$h(k_2, 0) = (90 \text{ mod } 10 + 0^2) \text{ mod } 10$

$= 0$

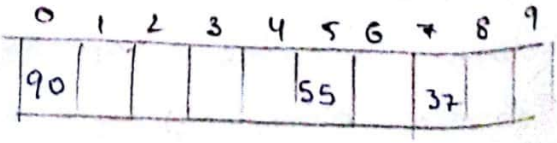


Since location 0 is empty, insert 90 in 0^{th} location.

Step 3 :

$h(k_3, 0) = (55 \text{ mod } 10) + 0^2) \text{ mod } 10$

$= 5$



Since location 5 is empty, insert 55 in 5^{th} location.

Step 4 :-

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

$H(K_4, 0) = (22 \bmod 10 + 0^2) \bmod 10 = 2$

Since 2nd location is empty insert 22 in location 2

Step 5 :-

$H(K_5, 0) = (17 \bmod 10 + 0^2) \bmod 10 = 7$

Since 7th location is not empty increment i

$H(K_5, 1) = (17 \bmod 10 + 1^2) \bmod 10$

$= 8$

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

Since 8th location is empty insert 17 in 8th location.

Step 6 :-

$H(K_6, 0) = (49 \bmod 10 + 0^2) \bmod 10$

$= 9$

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

Since 9th location is empty insert 49 in 9th location.

Step 7 :-

$h(K_7, 0) = (87 \bmod 10 + 0^2) \bmod 10 = 7$

$h(K_7, 1) = (87 \bmod 10 + 1^2) \bmod 10 = 8$

$h(K_7, 2) = (87 \bmod 10 + 2^2) \bmod 10$

$= 1$

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

Since 1st location is empty, insert 87 in 1st location.

To start with Double hashing using one hash value and then repeatedly steps forward an internal until on empty location is reached.

→ The internal is decided using a second independent hash function hence the name is called double hashing.

→ In double hashing we use two hash function rather than a single function.

→ The hash function in the case of double hashing can

be given as

$$h(k,i) = (h_1(k) + i\,h_2(k)) \bmod m$$

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

where m is size of the hash table

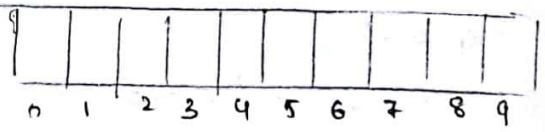
$h_1(k)$, $h_2(k)$ are two hash function

i is probe values that varies 0 to m-1

Eg :- Consider a hash table of size 10. Using double hashing insert the keys 72 27 36 24 63 81 92 101 into the table $h_1 = k \bmod 10$

$h_2 = k \bmod 8$

Initially the hash table with buckets are filled with -1.



Step 1 :

$h(k,i) = ((72 \% 10) + 0(72 \% 8)) \% 10$

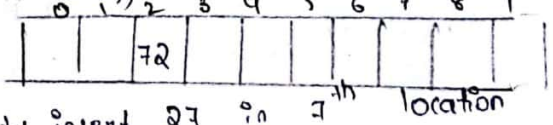
$= (2+0) \% 10 = 2$

Since 2nd location is empty insert 72 in 2nd location

Step 2 :

$h(k,8) = ((27 \% 10) + 0(27 \% 8)) \% 10$

$= 7$

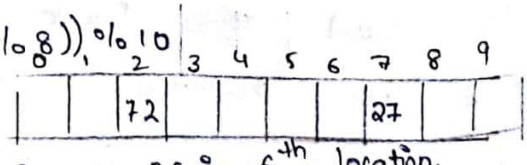


Since 7th location is empty. insert 27 in 7th location

Step 3 :

$h(k,0) = ((36 \% 10) + 0(36 \% 8)) \% 10$

$= 6$

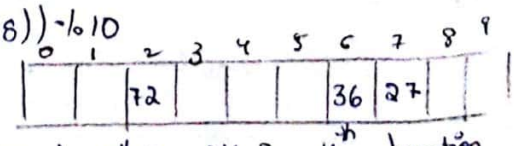


Since 6th location is empty, insert 36 in 6th location.

Step 4 :

$h(k,0) = ((24 \% 10) + 0(24 \% 8)) \% 10$

$= 4$



Since 4th location is empty, insert the 24 in 4th location

Step 5 :

$h(k,0) = ((63 \% 10) + 0\,(63 \% 8)) \% 10$

$= 3$

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

Since 3rd location is empty insert 63 in 3rd location

Step 6 :

$h(k,0) = ((81 \% 10) + 0\,(81 \% 8)) \% 10$

$= 1$

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

Since 1st location is empty, insert 81 in 1st location.

Step 7 :

$h(k,0) = ((92 \% 10) + 0\,(92 \% 8)) \% 10 = 2$

$h(k,1) = ((92 \% 10) + 1\,(92 \% 8)) \% 10 = 6$

$h(k,2) = ((92 \% 10) + 2\,(92 \% 8)) \% 10$

$= 0$

Since 0th location is empty insert 92 in 0th location.

Step 8 :

$h(k,0) = ((101 \% 10) + 0\,(101 \% 8)) \% 10$

$= 1$

$h(k,1) = ((101 \% 10) + 1\,(101 \% 8)) \% 10$

$= (1+5) \% 10$

$= 6$

$h(k,2) = ((101 \% 10) + 2\,(101 \% 8)) \% 10$

$= (1+10) \% 10$

$= 1$

$h(k,3) = ((101 \% 10) + 3\,(101 \% 8)) \% 10$

$= (1+15) \% 10$

$= 6$

$h(k,5) = ((101 \% 10) + 4\,(101 \% 8)) \% 10$

$= (1+25) \% 10$

$= 6$

$h(k,6) = ((101 \% 10) + 6\ (101 \% 8)) \% 10$

$= (1 + 30) \% 10$

$= 1$

$h(k,7) = ((101 \% 10) + 7\ (101 \% 8)) \% 10$

$= (1 + 35) \% 10$

$= 6$

Now location 1 & 6 is occupied so we can not store the key value 101. So Try again for the next location with probe number $i = 2$. Repeat the entire process until it doesnot count empty location.

Although double hashing algorithm, it always require M value to be prime number, in our case $m = 10$ so which is not prime number. Hence, M value have been equal to 11 the algorithm would be worked very efficiently.

RE-HASHING :

Re-hashing is a collision resolution technique in which the table is re-sized i.e., the size of table is doubled by creating a new table. It is preferable the total size of table is prime number.

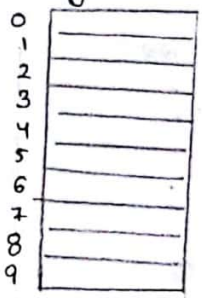
There are situations in which the re-hashing is required

→ When table is completely full

→ With double hashing when the table is filled half

→ When insertion failed due to overflow.

In such situations we have to transform key values from old table to new table by re computing their positions using suitable hash functions.



Consider we have to insert values 37, 19, 55, 22, 17, 49, 87 and 41, 56, 44, 43 hash table size is 10 & hash function is H(key) = key mod tablesize

0	90
1	41
2	22
3	43
4	44
5	55
6	56
7	37
8	17
9	49

S-1 ⇒ H(37) =
((37 %₀ 10) + 0) %₀ 10
= 7

S-2 ⇒ H(90) = (90 %₀ 10) + 0) %₀ 10
= 0

S-3 ⇒ H(55) = ((55 %₀ 10) + 0) %₀ 10
= 5

S-4 ⇒ H(22) = ((22 %₀ 10) + 0) %₀ 10 = 2

S-5 ⇒ H(17) = ((17 %₀ 10) + 0) %₀ 10 = 7
((17 %₀ 10) + 1) %₀ 10 =
7 + 1 = 8 %₀ 10 = 8

S-6 ⇒ H(49) = ((49 %₀ 10) + 0) %₀ 10
= 9

S-7 ⇒ H(41) = ((41 %₀ 10) + 0) %₀ 10
= 1

S-8 ⇒ H(56) = ((56 %₀ 10) + 0) %₀ 10
= 6

S-9 ⇒ H(44) = ((44 %₀ 10) + 0) %₀ 10
= 4

S-10 ⇒ H(43) = ((43 %₀ 10) + 0) %₀ 10
= 3

S-11 ⇒ H(87) = ((87 %₀ 10) + 0) %₀ 10
= 7

34
23
14

23 46 69 92

Hence we will re-hash the table by double the table size becomes table size is 20. But 20 is not a prime number we will prefer to make table size is 23

S-1 ⇒ H(37) =
= ((37 %₀ 23) + 0) %₀ 10) = 14

$H(98) = ((98 \% 23) + 0) \% 23)$

$= 21$

$H(55) = ((55 \% 23) + 0) \% 23) = 9$

$H(22) = ((22 \% 23) + 0) \% 23) = 22$

$H(17) = ((17 \% 23) + 0) \% 23) = 17$

$H(49) = ((49 \% 23) + 0) \% 23) = 3$

$H(41) = ((41 \% 23) + 0) \% 23) = 18$

$H(56) = ((56 \% 23) + 0) \% 23) = 10$

$H(40) = ((40 \% 23) + 0) \% 23) = 17$

$H(43) = ((43 \% 23) + 0) \% 23) = 20$

$H(87) = ((87 \% 23) + 0) \% 23) = 18$

$((87 \% 23) + 1) \% 23) = 19$

0	43
1	87
2	
3	49
4	
5	
6	
7	
8	
9	55
10	56
11	
12	
13	
14	37
15	
16	
17	17
18	41
19	40
20	43
21	98
22	22

Dynamic Hashing :-

To ensure good performance, it is necessary to increase the size of hash table, whenever it is loading density exceeds pre-specified threshold.

For example :- If we have currently 'B' buckets in hash table and using division hash function $h(k) = k \bmod size$. Then insert the loading density to exceed the pre specified threshold. We use array doubling to increase the number of buckets to $(2B+1)$ at the same time the hash function division changes to $2B+1$. This change in divisor requires used to rebuilt the hash table. The objective of dynamic hashing is to provide accept hashtable performance on per basic operations.

Dynamic Hashing is mainly divided into 2 types :-

Dynamic Hashing using Directories (Extendable Hashing)

Dynamic Hashing of Directories (Linear Hashing)

For Both forms we use the Hash function (H) that maps keys into non-negative integers. The range of H is assumed to be sufficiently large and we used to $H(k, p)$ to denote the integers formed by the 'p' least significant bits $H(k)$.

Ex :- A = 100, B = 101, C = 110 (0-7) bit representation Hash key s

A_0, A_1, B_0, B_1, C_1, C_2, C_3, C_5

key	Binary No.
A_0	100 000
A_1	100 001
B_0	101 000
B_1	101 001
C_1	110 001
C_2	110 010
C_3	110 011
C_5	110 101

Dynamic Hashing using Directories :-

We employee a directory 'd' of pointers to the buckets. The size of directory depends upon the no. of bits of H(k) used to index into the directory.

→ When indexing is done using H(k, 2) the directory size is $2^2 = 4$

→ When indexing is done using H(k, 5) the directory size is $2^5 = 32$

→ The no. of bits of H(k) used to index the directory is called the direcory depth.

→ The size of directory is 2^t where t is the directory depth & the number is almost equal to the directory size.

EXAMPLE :-

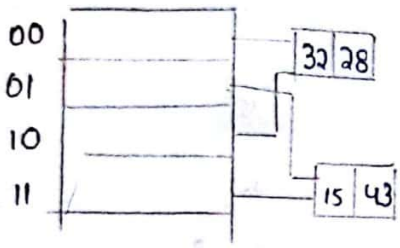
Insert the key values 32, 28, 43, 15, 66, 27, 86, 57, 35, 96, 72 in a Hash Table using directories using Dynamic Hashing.

key values are converted into binary values using same Hash funtion i.e., key mod 10.

Hash key	Hash function	Binary no.
32	32 % 10 = 2	0010
28	28 % 10 = 8	1000
43	43 % 10 = 3	0011
15	15 % 10 = 5	0101
66	66 % 10 = 6	0110
27	27 % 10 = 7	0111

86	86 % 10 = 6	0110
57	57 % 10 = 7	0111
35	35 % 10 = 5	0101
98	98 % 10 = 8	1000
72	72 % 10 = 2	0010

Dynamic Hash Table was directory depth is $2 = 2^2 = 4$ and each bucket have 2 slots (00, 01, 10, 11)



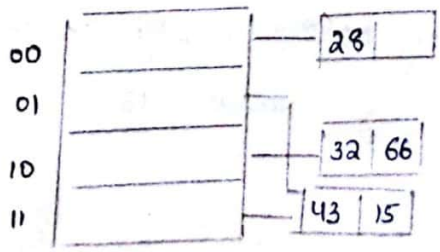
S-1 :- Insert the key value 32 (00 10) last two digits is 10. So, directory 10 is empty. So. insert, the key value 32 in bucket 0

S-2 :- Insert the key value 28 (1000) last two digits is 00. So. insert the key value 28 in bucket 0.

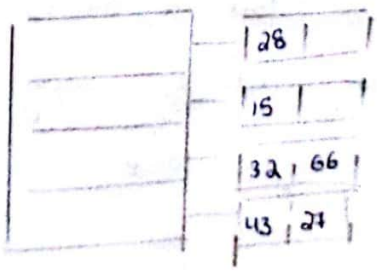
S-3 :- Insert the key value 43 (0011) last two digit is 11. So. directory 11 is empty. insert the key value 43 in bucket 1

S-4 :- Insert the key value 15 (0101). last two digits 01. So. directory 01 is empty. So. insert the key value 15 in bucket 1

S-5 :- Insert the key value 66 (0110). last 2 digits is 10. So, directory 10 is overflow. So, seperate the 10 bucket.

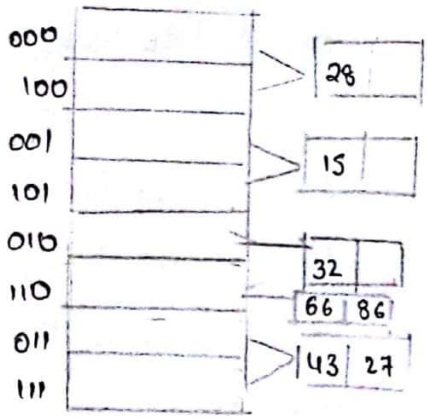


S-6 :- Insert the key value 27 (0111). last 2 digits 11. So, directory 11 is overflow. So, seperate the 11 bucket.



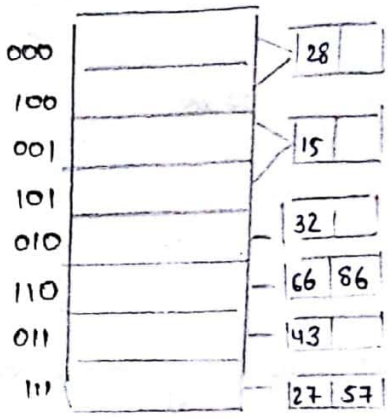
S-7 : Insert the key value 86 (0110) last 2 digits 10. So, directory 10 is overflow. So we use the directory depth is $2^3 = 8$

(000, 100, 001, 101, 010, 110, 111)

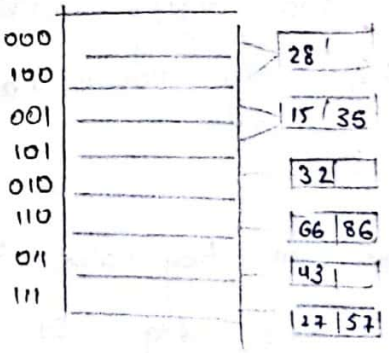


S-8 : Insert the key value 57 (011) last 3 digit 111.

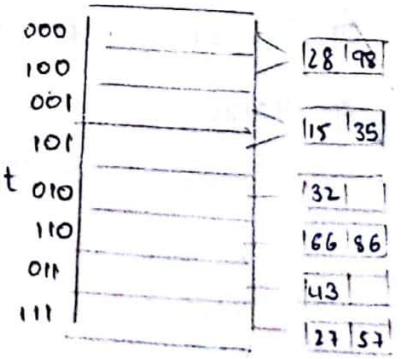
So, directory 111 is overflow. So, seperate the 111 bucket.



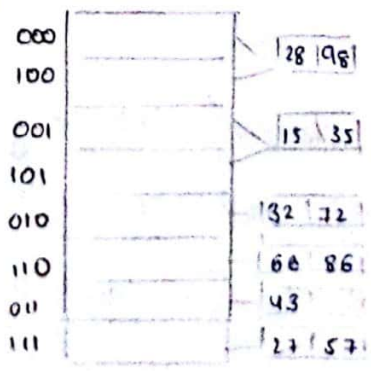
S-9 : Insert the key value 35 (0101). last 2 digit 01. So, directory 01 is empty. so insert the key value 35.



S-10 : Insert the key value 98 (1000) last 2 digits 00. So, directory 00 is empty. so insert the key value 98



S-11 : Insert the key value 72. (0010) last 3 digits 010. So, directory 010 is empty. So insert the key value 72.



DIRECTORLESS DYNAMIC HASHING :

→ This method is also called as linear dynamic hashing. Here we an array H(t) of buckets B is used. We assume that this array is as large as possible and so there is no possibility to increase in the size dynamically.

→ To avoid such large array we use to 2 variable Q & R ($0 \le q \le 2^r$) to keep track of the active bucket. R is number of bits of H(K) used to index into the Hash table. Q is the bucket that will split the next bucket

→ At any time only buckets 0 through $2^r + q - 1$ are active. Each active bucket is start of the chain of buckets. The remaining buckets on a chain are called overflow buckets.

→ Each dictionary pair is active of overflow buckets.

For example : hash table H(t) and key values $A_0, A_1, B_4, B_5, C_2, C_3, C_5, C_1$ and $A_0 = 100$, $B = 101$, $C = 110$. Inserting key values in a hash table using directorless dynamic hashing.

A_0	100	000
A_1	100	001
B_4	101	100
B_5	101	101
C_2	110	00
C_3	110	011
C_5	110	101
C_1	110	001

The number of active buckets is 4 00, 01, 10, 11 and each bucket have 2 slots

① :

00	B_4, A_0
01	A_1, B_5
10	C_2
11	C_3

② :

000	A_0
001	A_1, A_5 → C_5
010	C_2
011	C_3
100	B_4

③ :

000	A_0
0010	A_1, C_1
010	C_2
011	C_3
100	B_4
101	B_5, C_5