UNIT V SORTING, SEARCHING AND HASH TECHNIQUES

Sorting algorithms: Insertion sort - Selection sort - Shell sort - Bubble sort - Quick sort - Merge sort - Radix sort – Searching: Linear search –Binary Search Hashing: Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing.

SORTING:

Definition:

Sorting is a technique for arranging data in a particular order.

Order of sorting:

Order means the arrangement of data. The sorting order can be ascending or descending. The ascending order means arranging the data in increasing order and descending order means arranging the data in decreasing order.

Types of Sorting

Internal Sorting
External Sorting

Internal Sorting

Internal Sorting is a type of sorting technique in which data resides on main memory of computer. It is applicable when the number of elements in the list is small.

E.g. Bubble Sort, Insertion Sort, Shell Sort, Quick Sort., Selection sort, Radix sort

External Sorting

External Sorting is a type of sorting technique in which there is a huge amount of data and it resides on secondary devise(for eg hard disk,Magnetic tape and so no) while sorting.

E.g. Merge Sort, Multiway Merge Sort,Polyphase merge sort

Sorting can be classified based on

1.Computational complexity

2.Memory utilization

3. Stability

4. Number of comparisons.

ANALYSIS OF ALGORITHMS:

Efficiency of an algorithm can be measured in terms of:

Space Complexity: Refers to the space required to execute the algorithm

Time Complexity: Refers to the time required to run the program.

Sorting algorithms:

- Insertion sort
- Selection sort
- Shell sort
- Bubble sort
- Quick sort
- Merge sort
- Radix sort

INSERTION SORTING:

- The insertion sort works by taking elements from the list one by one and inserting them in the correct position into a new sorted list.
- Insertion sort consists of N-1 passes, where N is the number of elements to be sorted.
- The i^{th} pass will insert the i^{th} element A[i] into its rightful place among A[1],A[2],…,A[i-1].
- After doing this insertion the elements occupying A[1],…A[i] are in sorted order.

How Insertion sort algorithm works?

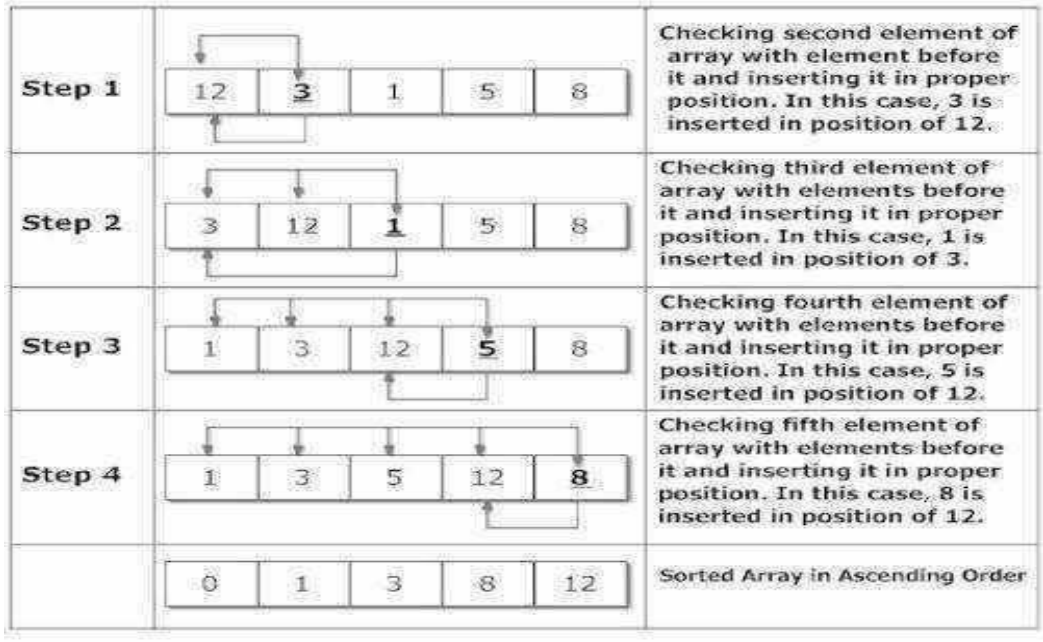


Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

Insertion Sort routine:

```
void Insertion_sort(int a[ ], int n)
{
          int i, j, temp;
          for ( i = 0 ; i < n -1 ; i ++ )
                    {
                    temp = a [ j ] ;


           for ( j = i ; j > 0 && a [ j -1 ] > temp ; j -- )
                   {
                    a[ j ] = a [ j – 1 ] ;

                    }
        a[j]=temp;
  }}
```

Program for Insertion sort

```
#include<stdio.h>
void main( ){
 int n, a[ 25 ], i, j, temp;
 printf( "Enter number of elements \n" );
     scanf( "%d", &n );
  printf( "Enter %d integers \n", n );
  for ( i = 0; i < n; i++ )
      scanf( "%d", &a[i] );
  for ( i = 0 ; i < n; i++ ){
temp=a[i];
for (j=i;j > 0 && a[ j -1]>temp;j--)
         {
              a[ j ]    = a[ j - 1 ];
         }
         a[j]=temp;}
 printf( "Sorted list in ascending order: \n ");
 for ( i = 0 ; i < n ; i++)
     printf ( "%d \n ", a[ i ] );}
```

OUTPUT:

```
Enter number of elements
6
Enter 6 integers
20 10 60 40 30 15
Sorted list in ascending order:
 10
 15
 20
 30
```

40
60

Advantage of Insertion sort

- Simple implementation.
- Efficient for (quite) small data sets.
- Efficient for data sets that are already substantially sorted.

Disadvantages of Insertion sort

- It is less efficient on list containing more number of elements.
- As the number of elements increases the performance of the program would be slow.
- Insertion sort needs a large number of element shifts.

Selection Sort

Selection sort selects the smallest element in the list and place it in the first position then selects the second smallest element and place it in the second position and it proceeds in the similar way until the entire list is sorted. For "n" elements, (n-1) passes are required. At the end of the i^{th} iteration, the i^{th} smallest element will be placed in its correct position.

Selection Sort routine:

```
void Selection_sort( int a[ ], int n )
{
    int i , j , temp , position ;
        for ( i = 0 ; i < n – 1 ; i ++ )
        {
                position = i ;
                for ( j = i + 1 ; j < n ; j ++ )
                {
                        if ( a[ position ] > a[ j ] )
                                position = j;}
                temp = a[ i ];
                a[ i ] = a[ position ];
                a[ position ] = temp;
        }}
```

How Selection sort algorithm works?

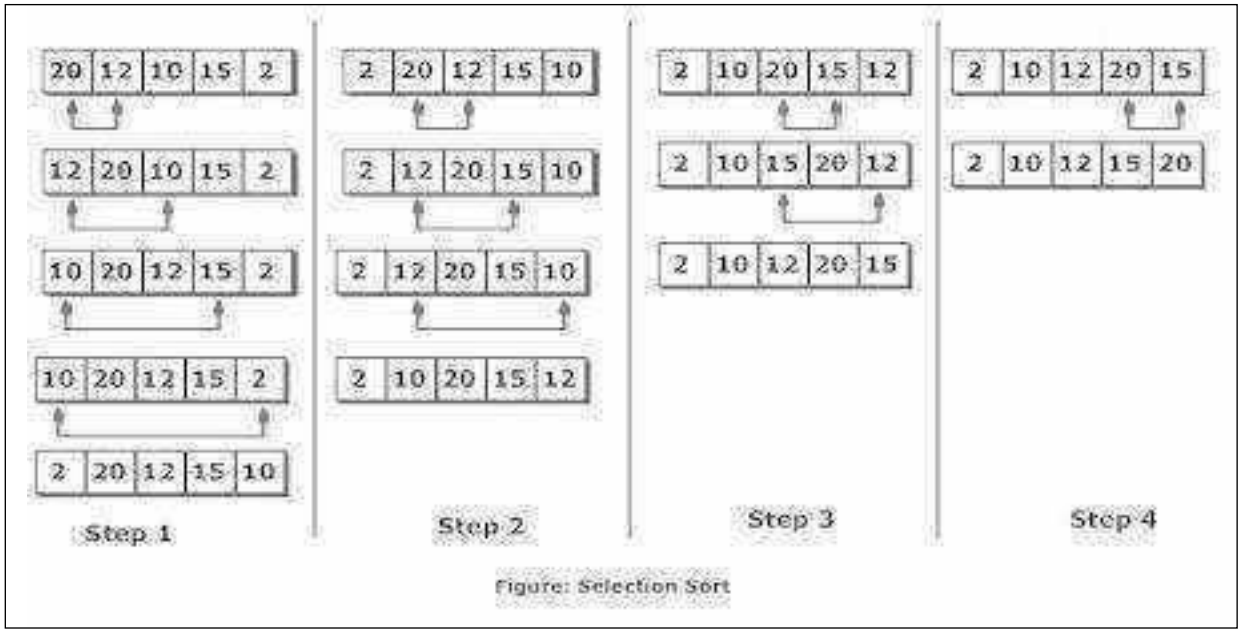


Figure: Selection Sort

Program for Selection sort

```
#include <stdio.h>
 void main( )
{
   int a [ 100 ] , n , i , j , position , temp ;
   printf ( "Enter number of elements \n" ) ;
       scanf ( "%d", &n ) ;
   printf ( " Enter %d integers \n ", n ) ;
   for ( i = 0 ; i < n ; i ++ )
      scanf ( "%d", & a[ i ] ) ;
   for ( i = 0 ; i < ( n - 1 ) ; i ++ )
     {
     position = i ;
     for ( j = i + 1 ; j < n ; j ++ )
     {
         if ( a [ position ] > a [ j ] )
         position = j ;
     }
    if ( position != i )
    {
      temp = a [ i ] ;
      a [ i ] = a [ position ] ;
      a [ position ] = temp ;
    }
  }
   printf ( "Sorted list in ascending order: \n ") ;
   for ( i = 0 ; i < n ; i ++ )
      printf ( " %d \n ", a[ i ] ) ;
 }
```

OUTPUT:

```
Enter number of elements
5
 Enter 5 integers
 8 3 9 5 1
Sorted list in ascending order:
  1
  3
  5
  8
  9
```

Advantages of selection sort

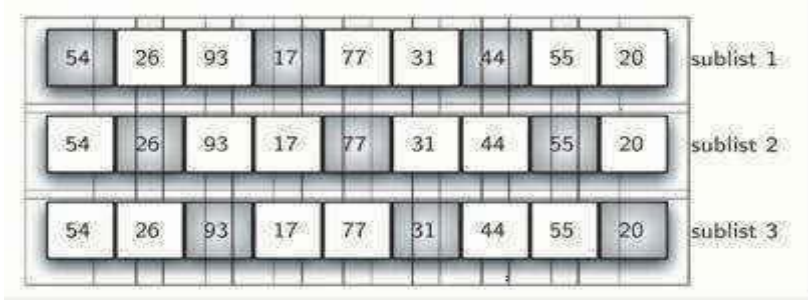
- Memory required is small.
- Selection sort is useful when you have limited memory available.
- Relatively efficient for small arrays.

Disadvantage of selection sort

- Poor efficiency when dealing with a huge list of items.
- The selection sort requires n-squared number of steps for sorting n elements.
- The selection sort is only suitable for a list of few elements that are in random order.

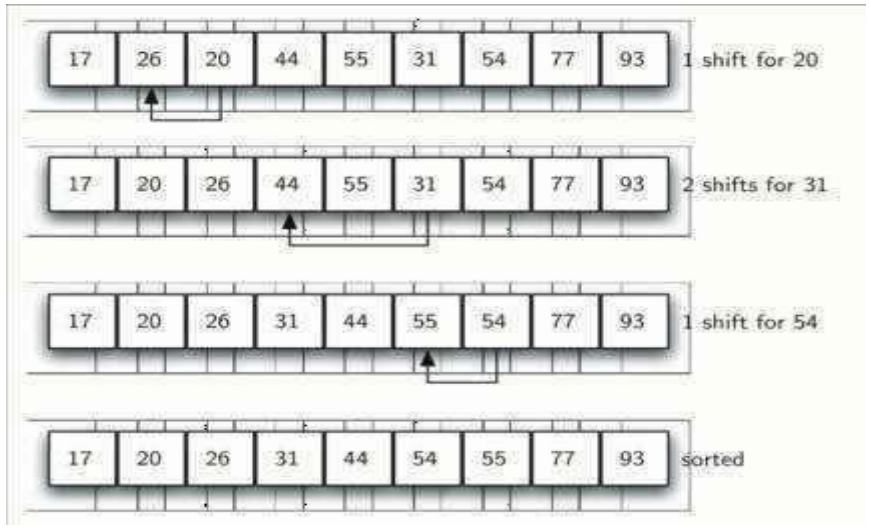
Shell Sort

- Invented by Donald shell.
- It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time.
- In shell sort the whole array is first fragmented into K segments, where K is preferably a prime number.
- After the first pass the whole array is partially sorted.
- In the next pass, the value of K is reduced which increases the size of each segment and reduces the number of segments.
- The next value of K is chosen so that it is relatively prime to its previous value.
- The process is repeated until K=1 at which the array is sorted.
- The insertion sort is applied to each segment so each successive segment is partially sorted.
- The shell sort is also called the Diminishing Increment sort, because the value of k decreases continuously



A Shell Sort with Increments of Three

A Shell Sort after Sorting Each Sublist



Shell Sort: A Final Insertion Sort with Increment of 1

Shell Sort routine:

```
void Shell_sort ( int a[ ], int n )
{
        int i, j, k, temp;
        for ( k = n / 2 ; k > 0 ; k = k / 2 )
            for ( i = k ; i < n ; i + + )
            {
            temp = a [ i ] ;
        for ( j = i ; j > = k && a [ j – k ] > temp ; j = j – k )
                        {
                            a [ j ] = a [ j – k ] ;
                        }
                    a [ j ] = temp ;
            }
}
```

Program for Shell sort

```
#include<stdio.h>
 void main( )
{
  int n, a[ 25 ], i, j,k,temp;
  printf( "Enter number of elements \n" );
  scanf( "%d", &n );
```

```
   printf( "Enter %d integers \n", n );
   for ( i = 0; i < n; i++ )
       scanf( "%d", &a[i] );
                        for (k = n / 2 ; k>0 ; k=k/ 2){


              for ( i = k ; i < n ; i ++ )
              {
                temp = a [ i ] ;
                    for (j = i ; j>= k && a [ j - k ]>temp ; j=j - k )
                                                          {
                                                                  a [ j ] = a [ j - k ] ;
                                                          }
                                                  a [ j ] = temp ;
              }
       }
 printf( "Sorted list in ascending order using shell sort: \n ");
  for ( i = 0 ; i < n ; i++)
       printf ( "%d\t ", a[ i ] );
  }
```

OUTPUT:

```
Enter number of elements
10
Enter 10 integers
81 94 11 96 12 35 17 95 28 58
Sorted list in ascending order using shell sort:
 11      12      17      28      35      58      81      94      95      96
```

//PROGRAM FOR SHELL USING FUNCTION

```
#include < stdio.h >
 void main( )
{
    int a [ 5 ] = { 4, 5, 2, 3, 6 } , i = 0 ;
    ShellSort ( a, 5 ) ;
printf("Example using function");
    printf( " After Sorting :" ) ;
    for ( i = 0 ; i < 5 ; i ++ )
        printf ( " %d ", a[ i ] ) ;
}
void ShellSort (int a [ 5 ] , int n )
{
        int i , j , k , temp ;
        for ( k = n / 2 ; k > 0 ; k / =  2)
        {
              for ( i = k ; i < n ; i ++ )
```

```
{
    temp = a [ i ] ;
   for ( j = i ; j > = k && a [ j – k ] > temp ; j = j – k ){
                                        a [ j ] = a [ j – k ] ;
                                }
                        a [ j ] = temp ; } }}
```

OUTPUT:

After Sorting : 2 3 4 5 6

Advantages of Shell sort

- Efficient for medium-size lists.

Disadvantages of Shell sort

- Complex algorithm, not nearly as efficient as the merge, heap and quick sorts

Bubble Sort

- Bubble sort is one of the simplest internal sorting algorithms.
- Bubble sort works by comparing two consecutive elements and the largest element among these two bubbles towards right at the end of the first pass the largest element gets sorted and placed at the end of the sorted list.
- This process is repeated for all pairs of elements until it moves the largest element to the end of the list in that iteration.
- Bubble sort consists of (n-1) passes, where n is the number of elements to be sorted.
- In 1st pass the largest element will be placed in the n^{th} position.
- In 2nd pass the second largest element will be placed in the $(n-1)^{th}$ position.
- In $(n-1)^{th}$ pass only the first two elements are compared.

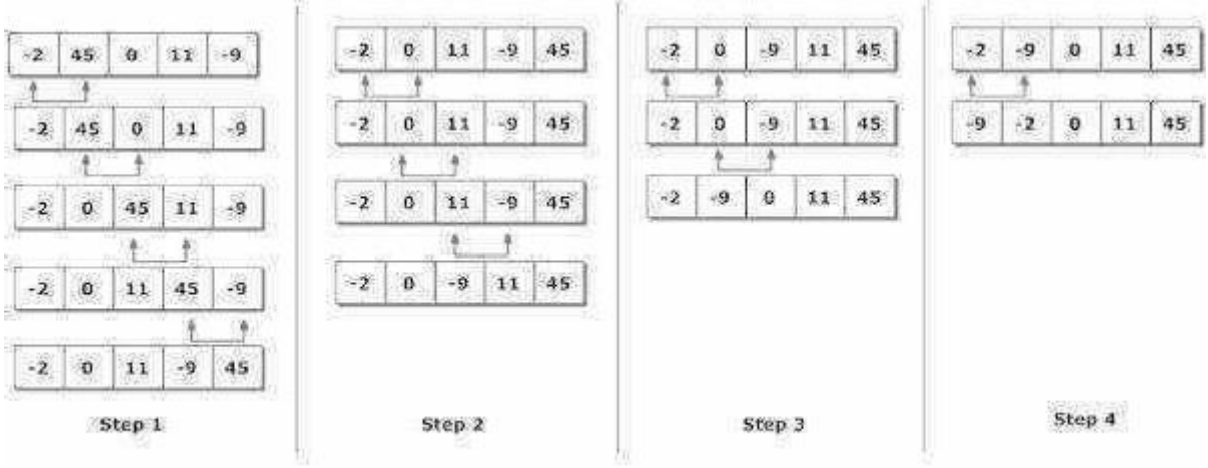


Figure: Working of Bubble sort algorithm

Bubble sort routine:

```
void Bubble_sort (int a [ ] , int n )
{
        int i, j, temp;
        for( i = 0; i < n - 1; i++ ){


            for( j = 0; j < n – i - 1; j++ )
            {
                    if( a[ j ] > a [ j + 1 ] )
                    {
                            temp = a [ j ];
                            a[ j ] = a[ j + 1 ];
                            a[ j + 1 ] = temp;
                    }
            } }}
```

Program for Bubble sort

```
#include<stdio.h >
#include<conio.h >
void main( )
{
int a [ 20 ], i, j, temp, n ;
printf ("Enter the number of elements");
    scanf ("%d",&n);
printf("Enter the numbers");
for(i=0;i < n ;i++)
scanf("%d",&a[i]);
for(i=0;i<n-1;i++)
{
for(j=0;j<n-i-1;j++)
{
if(a[j]>a[j + 1]){
                    temp = a[ j ] ;
                    a[ j ] = a[ j + 1 ] ;
        a[ j+ 1 ] = temp;
    }
}
}
printf("\nSorted array\t");
for(i=0;i<n;i++)
    printf("%d\t",a[i]);
}
```

OUTPUT:

Enter the number of elements5

Enter the numbers8 3 9 5 1

Sorted array 1 3 5 8 9

Advantage of Bubble sort

- It is simple to write
- Easy to understand
- It only takes a few lines of code.

Disadvantage of Bubble sort

- The major drawback is the amount of time it takes to sort.
- The average time increases almost exponentially as the number of table elements increase.

Quick Sort

Quicksort is a divide and conquer algorithm.

The basic idea is to find a "pivot" item in the array and compare all other items with pivot element.

Shift items such that all of the items before the pivot are less than the pivot value and all the items after the pivot are greater than the pivot value.

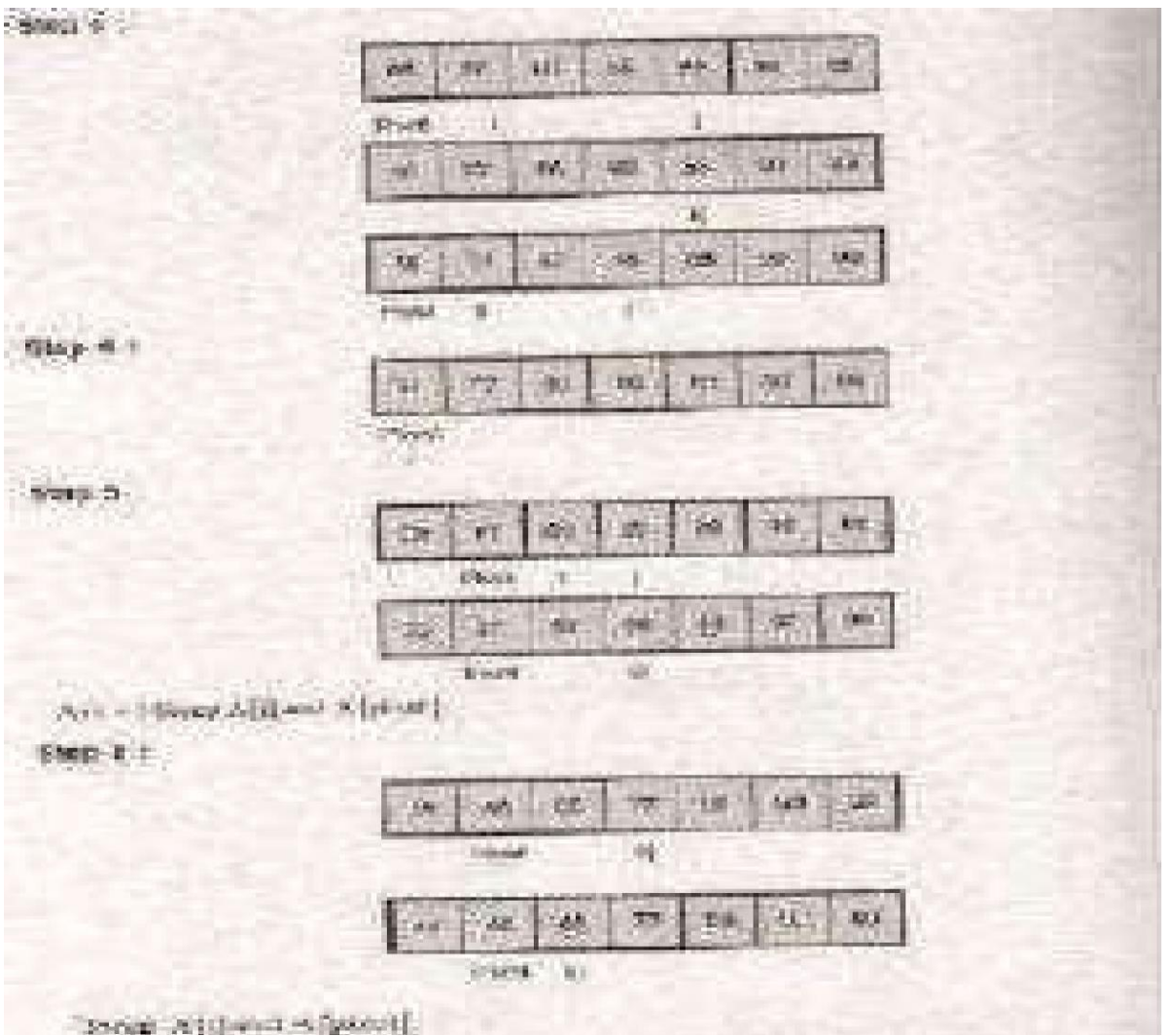
After that, recursively perform the same operation on the items before and after the pivot.

Find a "pivot" item in the array. This item is the basis for comparison for a single round.

Start a pointer (the left pointer) at the first item in the array.

Start a pointer (the right pointer) at the last item in the array.

1. Assume A[0]=pivot which is the left. i.e pivot=left.
2. Set i=left+1; i.e A[1];
3. Set j=right. ie. A[6] if there are 7 elements in the array
4. If A[pivot]>A[i],increment i and if A[j]>A[pivot],then decrement j, Otherwise swap A[i] and A[j] element.
5. If i=j,then swap A[pivot] and A[j].



Quick Sort routine:

```
void Quicksort ( int a [ ], int left, int right )
{
        int i, j, p, temp;
        if ( left < right )
        {
                p = left;
                i = left + 1; j
                =  right;
                while ( i < j )
                {
                        while ( a [ i ] < = a [ p ] )
                                i = i + 1;
                        while ( a [ j ] > a [ p ] )
                                j = j - 1;
                        if ( i < j )
                        {
                                temp = a [ i ];
                                a [ i ] = a [ j ];
                                a [ j ] = temp;
                        }
                }
                temp = a [ p ];
                a [ p ] = a [ j ];
                a [ j ] = temp;
                quicksort ( a, left, j - 1 );
                quicksort ( a, j + 1, right );
        } }
```

Program for Quick sort

```
#include<stdio.h>
void quicksort (int [10], int, int ) ;
void main( )
{
  int a[20], n, i ;
  printf("Enter size of the array:" );
      scanf("%d",&n);
          printf( " Enter the numbers :");
          for ( i = 0 ; i < n ; i ++ )
             scanf ("%d",&a[i]);
         quicksort ( a , 0 , n - 1 );
         printf ( " Sorted elements: " );
         for ( i = 0 ; i < n ; i ++ )
         printf ("%d\t",a[ i]);
}
 void quicksort ( int a[10], int left, int right )
```

```
{
 int p, j, temp, i ;
 if ( left < right )
 {
  p = left  ;
  i =  left  ;
  j = right ;
  while ( i < j )
  {
    while(a[i]<= a[p] && i<right )
          i++ ;
    while ( a [ j ] > a [ p ] )
          j--;
    if ( i < j )
    {
          temp = a [ i ]  ;
          a [ i ] = a [ j ] ;
          a[ j ] = temp ;
    }
  }
          temp  =  a [ p ] ;
          a [ p ] = a [ j ]  ;
          a [ j ] =temp ;
  quicksort ( a , left , j - 1 ) ;
  quicksort ( a , j + 1 , right ) ;
}
}
```

OUTPUT:

```
Enter size of the array:8

 Enter the numbers :40 20 70 14 60 61 97 30

 Sorted elements: 14    20      30      40      60      61      70      97
```

Advantages of Quick sort

- Fast and efficient as it deals well with a huge list of items.
- No additional storage is required.

Disadvantages of Quick sort

- The difficulty of implementing the partitioning algorithm.

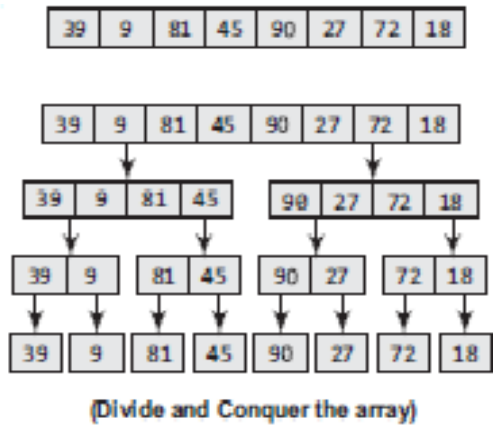
Merge Sort

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.

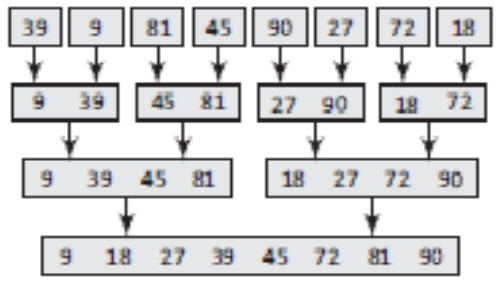
If there are more elements in the array, divide A into two sub-arrays, A1 and A2, each containing about half of the elements of A.
Conquer means sorting the two sub-arrays recursively using merge sort.
Combine means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.

The basic steps of a merge sort algorithm are as follows:

- If the array is of length 0 or 1, then it is already sorted.
- Otherwise, divide the unsorted array into two sub-arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array.
- Merge the two sub-arrays to form a single sorted list.



(Divide and Conquer the array)



(Combine the elements to form a sorted array)

Merge Sort routine:

```
void Merge_sort (int a [ ] , int temp [ ] , int n )
 {
      msort ( a , temp , 0 , n - 1 ) ;
 }
        int center ;
        if( left < right ){
                center = ( left + right ) / 2 ;
                msort ( a , left , center ) ;
                msort ( a , temp , center + 1 , right ) ;
```

```
                    merge ( a , temp , n , left , center , right ) ;
                }}
void merge ( int a [ ] , int temp [ ] , int n , int left , int center , int right )
{
        int i = 0 , j , left_end = center , center = center + 1 ;
        while( ( left < = left_end ) && ( center < = right ) )
        {
                if( a [ left ] < = a [ center ] )
                {
                        temp [ i ] = a [ left ] ;
                        i + + ;
                        left + + ;
                }
                else
                {
                        temp [ i ] = a [ center ] ;
                        i + + ;
                        center + + ;
                }
        }
        while( left <= left_end )
        {
                temp [ I ] = a [ left ] ;
                left + + ;
                i + + ;
        }
        while( center < = right )
        {
                temp [ i ] = a [ center ] ;
                center + + ;
                i + + ;
        }
        for ( i = 0 ; i < n ; i + + )
             print temp [ i ] ;
}
```

Program for merge sort

```
#include<stdio.h>
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);
```

```
int main()
{
   int a[30],n,i;
   printf("Enter no of elements:");
   scanf("%d",&n);
   printf("Enter array elements:");
   for(i=0;i<n;i++)
   scanf("%d",&a[i]);
   mergesort(a,0,n-1);
   printf("\nSorted array is :");
   for(i=0;i<n;i++)
   printf("%d ",a[i]);
   return 0;
}
void mergesort(int a[],int i,int j)
{
   int mid;
   if(i<j)
   {
      mid=(i+j)/2;
      mergesort(a,i,mid);        //left recursion mergesort(a,mid+1,j);
      //right recursion merge(a,i,mid,mid+1,j);    //merging of two
      sorted sub-arrays
   }}
void merge(int a[],int i1,int j1,int i2,int j2)
{
   int temp[50];    //array used for merging
   int i,j,k;
   i=i1;    //beginning of the first list
   j=i2;    //beginning of the second list
   k=0;
   while(i<=j1 && j<=j2)    //while elements in both lists
   { if(a[i]<a[j])
   temp[k++]=a[i++];
   else
   temp[k++]=a[j++];
   }
   while(i<=j1)    //copy remaining elements of the first list
   temp[k++]=a[i++];
   while(j<=j2)    //copy remaining elements of the second list
   temp[k++]=a[j++];
   //Transfer elements from temp[] back to a[]
   for(i=i1,j=0;i<=j2;i++,j++)
      a[i]=temp[j];
}
```

OUTPUT:

Enter no of elements:8
Enter array elements:24 13 26 1 2 27 38 15
Sorted array is :1 2 13 15 24 26 27 38

Advantages of Merge sort

- Mergesort is well-suited for sorting really huge amounts of data that does not fit into memory.
- It is fast and stable algorithm

Disadvantages of Merge sort

- Merge sort uses a lot of memory.
- It uses extra space proportional to number of element n.
- This can slow it down when attempting to sort very large data.

Radix Sort

Radix sort is one of the linear sorting algorithms. It is generalized form of bucket sort. It can be performed using buckets from 0 to 9.
It is also called binsort, card sort.
It works by sorting the input based on each digit. In first pass all the elements are stored according to the least significant digit.
In second pass the elements are arranged according to the next least significant digit and so on till the most significant digit.
The number of passes in a Radix sort depends upon the number of digits in the given numbers.

Algorithm for Radix sort

Steps1: Consider 10 buckets (1 for each digit 0 to 9)

Step2: Consider the LSB (Least Significant Bit) of each number (numbers in the one‟s

Place…. E.g., in 43 LSB = 3)

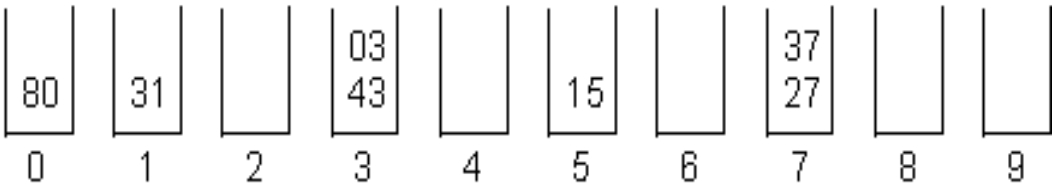
Step3: Place the elements in their respective buckets according to the LSB of each number

Step4: Write the numbers from the bucket (0 to 9) bottom to top.

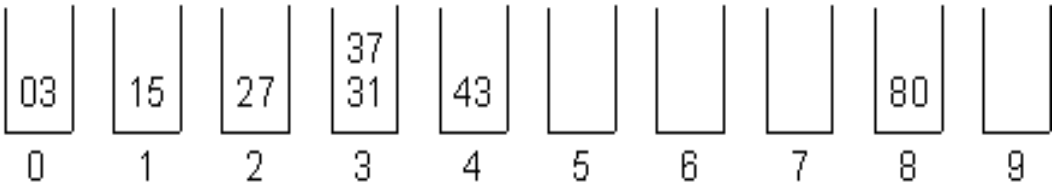
Step5: repeat the same process with the digits in the 10‟s place (e.g. In 43 MSB =4)

Step6: repeat the same step till all the digits of the given number are consider.

43 27 31 15 37 80 03



80 31 43 03 15 27 37



03 15 27 31 37 43 80

Sorted list of array : 3 15 27 31 37 43 80

Routine for Radix sort

```
void Radix_sort ( int a [ ] , int n )
{
        int bucket [ 10 ] [ 5 ] , buck [ 10 ] , b [ 10 ] ;
        int i , j , k , l , num , div , large , passes ;
        div = 1 ;
        num = 0 ;
        large = a [ 0 ] ;
        for ( i = 0 ; i < n ; i ++ )
        {
                if ( a[ I ] > large )
                {
                        large = a [ i ] ;
                }
                while ( large > 0 )
                {
                        num ++ ;
```

```
                large = large / 10 ;
        }
        for ( passes = 0 ; passes < num ; passes ++ )
        {
                for ( k = 0 ; k < 10 ; k ++ )
                {
                        buck [ k ] = 0 ;
                }
                for ( i = 0 ; i < n ; i ++ )
                {
                        l = ( ( a [ i ] / div ) % 10 ) ;
                        bucket [ l ] [ buck [ l ] ++ ] = a [ i ] ;
                }
                i = 0 ;
                for ( k = 0 ; k < 10 ; k ++ )
                {
                        for ( j = 0 ; j < buck [ k ] ; j ++ )
                        {
                                a [ i ++ ] = bucket [ k ] [ j ] ;
                        }
                }
                div * = 10 ;
        }
    }
}
```

Program for Radix sort

```
#include<stdio.h >
 void main( )
{
   int a [ 5 ] = { 4, 5, 2, 3, 6 } , i = 0 ;
        void Radix_sort ( int a [ ] , int n );
        Radix_sort(a,5);
   printf( " After Sorting :" ) ;
   for ( i = 0 ; i < 5 ; i ++ )
      printf ( " %d ", a[ i ] ) ;
}
void Radix_sort ( int a [ ] , int n )
{
        int bucket [ 10 ] [ 5 ] , buck [ 10 ] , b [ 10 ] ;
        int i , j , k , l, num , div , large , passes ;
        div = 1 ;
        num = 0 ;
        large = a [ 0 ] ;
        for ( i = 0 ; i < n ; i ++ ){
```

```
            if ( a[ i] > large )
            {
                large = a [ i ] ;
            }
            while ( large > 0 )
            {
                num ++ ;
                large = large / 10 ;
            }
            for ( passes = 0 ; passes < num ; passes ++ )
            {
                for ( k = 0 ; k < 10 ; k ++ )
                {
                    buck [ k ] = 0 ;
                }
                for ( i = 0 ; i < n ; i ++ )
                {
                    l = ( ( a [ i ] / div ) % 10 ) ;
                    bucket [ l ] [ buck [ l ] ++ ] = a [ i ] ;
                }
                i = 0 ;
                for ( k = 0 ; k < 10 ; k ++ )
                {
                    for(j=0 ; j<buck[k];j++ )
                    {
                        a[i++]=bucket[ k ][ j ] ;
                    }
                }
                div*= 10 ;
            }
        }
}
```

Advantages of Radix sort:

- Fast and complexity does not depend on the number of data.
- Radix Sort is very simple.

Disadvantages of Radix sort:

- Radix Sort takes more space than other sorting algorithms, since in addition to the array that will be sorted, you need to have a sub list for each of the possible digits or letters.
- Since Radix Sort depends on the digits or letters, Radix Sort is also much less flexible than other sorts.

Analysis of Sorting algorithms:

S.No	Algorithm	Best Case Analysis	Average Case Analysis	Worst Case Analysis
1	Insertion sort	O(N)	$O(N^2)$	$O(N^2)$
2	Selection sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
3	Shell sort	O(N log N)	$O(N^{1.5})$	$O(N^2)$
4	Bubble sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
5	Quick sort	O(N log N)	O(N log N)	$O(N^2)$
6	Merge sort	O(N log N)	O(N log N)	O(N log N)
7	Radix or bucket or binsort sort or card sort	O(N log N)	O(N log N)	O(N log N)

SEARCHING

Searching is an algorithm, to check whether a particular element is present in the list.

Types of searching:-

- Linear search
- Binary Search

Linear Search

Linear search is used to search a data item in the given set in the sequential manner, starting from the first element. It is also called as sequential search

Linear Search routine:

```
void Linear_search ( int a[ ] , int n )
{
int search , count = 0 ;
for ( i = 0 ; i < n ; I ++ )
{
if ( a [ i ] = = search )
{
count ++ ;
}
}
if ( count = = 0 )
print “Element not Present” ;
else
print “Element is Present in list" ;
}
```

Program for Linear search

```
#include < stdio.h >
void main( )
{
int a [ 10 ] , n , i , search, count = 0 ;
printf ( " Enter the number of elements \ t " ) ;
   scanf ( " %d " , & n ) ;
printf ( " \n Enter %d numbers \n " , n ) ;
for ( i = 0 ; i < n ; i ++ )
   scanf ( " %d " , & a [ i ] ) ;
printf ( " \n Array Elements \n " ) ;
for ( i = 0 ; i < n ; i ++ )
    printf ( " %d \t " , a [ i ] ) ;
printf ( " \ n \ n Enter the Element to be searched: \ t " ) ;
   scanf ( " % d " , & search ) ;
for ( i =0 ; i < n; i ++ )
  {
    if ( search = = a [ i ] )

      count ++ ;
 }
if ( count = = 0 )
  printf( " \n Element %d is not present in the array " , search ) ;
else
  printf ( " \n Element %d is present %d times in the array \n " , search , count ) ;
}
```

OUTPUT:

```
Enter the number of elements    5
Enter the numbers
20 10 5 25 100
Array Elements
20      10      5       25      100
Enter the Element to be searched:       25
Element 25 is present 1 times in the array
```

Advantages of Linear search:

- The linear search is simple - It is very easy to understand and implement;
- It does not require the data in the array to be stored in any particular order.

Disadvantages of Linear search:

- Slower than many other search algorithms.

- It has a very poor efficiency.

Binary Search

- Binary search is used to search an item in a sorted list. In this method , initialize the lower limit and upper limit.
- The middle position is computed as (first+last)/2 and check the element in the middle position with the data item to be searched.
- If the data item is greater than the middle value then the lower limit is adjusted to one greater than the middle value.Otherwise the upper limit is adjusted to one less than the middle value.

Working principle:

Algorithm is quite simple. It can be done either recursively or iteratively:

1. Get the middle element;
2. If the middle element equals to the searched value, the algorithm stops;
3. Otherwise, two cases are possible:
    - Search value is less than the middle element. In this case, go to the step 1 for the part of the array, before middle element.
    - Searched value is greater, than the middle element. In this case, go to the step 1 for the part of the array, after middle element.

Example 1.

Find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 (middle element is 19 > 6): -1 5 6 18 19 25 46 78 102 114

Step 2 (middle element is 5 < 6): -1 5 6 18 19 25 46 78 102 114

Step 3 (middle element is 6 == 6): -1 5 6 18 19 25 46 78 102 114

Binary Search routine:

```
void Binary_search ( int a[ ] , int n , int search )
{
int first, last, mid ;
first = 0 ;
last = n-1 ;
mid = ( first + last ) / 2 ;
while ( first < = last )
{
if ( Search > a [ mid ] )
first = mid + 1 ;
else if ( Search = = a [ mid ] )
{
print “Element is present in the list" ;
break ;
}
```

```
else {


last = mid – 1 ;
mid = ( first + last ) / 2 ;
}
if( first > last )
print “Element Not Found” ;
}
```

Program for Binary Search:

```
#include<stdio.h>
void main( )
{
int a [ 10 ] , n , i , search, count = 0 ;
void Binary_search ( int a[ ] , int n , int search );
printf ("Enter the number of elements \t") ;
   scanf ("%d",&n);
printf("\nEnter the numbers\n") ;
for (i = 0; i<n;i++)
   scanf("%d",&a[i]);
printf("\nArray Elements\n") ;
for (i = 0 ; i < n ; i ++ )
printf("%d\t",a[i]) ;
printf ("\n\nEnter the Element to be searched:\t");
scanf("%d",&search  );
Binary_search(a,n,search);
}
void Binary_search ( int a[ ] , int n , int search )
{
int first, last, mid ;
first = 0 ;
last = n-1 ;
mid = (first + last ) / 2 ;
while (first<=last )
{
if(search>a[mid])
first = mid + 1 ;
else if (search==a[mid])
{
printf("Element is present in the list");
break ;
}
else
last = mid - 1 ;
mid = ( first + last ) / 2 ;
}
if( first > last )
printf("Element Not Found");
```

}

OUTPUT:

Enter the number of elements 5

Enter the numbers

20 25 50 75 100

Array Elements

20 25 50 75 100

Enter the Element to be searched: 75

Element is present in the listPress any key to continue . . .

Advantages of Binary search:

- In Linear search, the search element is compared with all the elements in the array. Whereas in Binary search, the search element is compared based on the middle element present in the array.
- A technique for searching an ordered list in which we first check the middle item and - based on that comparison - "discard" half the data. The same procedure is then applied to the remaining half until a match is found or there are no more items left.

Disadvantages of Binary search:

- Binary search algorithm employs recursive approach and this approach requires more stack space.
- It requires the data in the array to be stored in sorted order.
- It involves additional complexity in computing the middle element of the array.

Analysis of Searching algorithms:

S.No	Algorithm	Best Case Analysis	Average Case Analysis	Worst Case Analysis
1	Linear search	O(1)	O(N)	O(N)
2	Binary search	O(1)	O(log N)	O(log N)

HASHING :

Hashing is a technique that is used to store, retrieve and find data in the data structure called Hash Table. It is used to overcome the drawback of Linear Search (Comparison) & Binary Search (Sorted order list). It involves two important concepts-

- Hash Table
- Hash Function

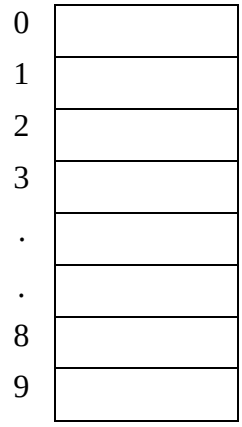
Hash table

- A **hash table** is a data structure that is used to store and retrieve data (keys) very quickly.
- It is an array of some fixed size, containing the keys.
- Hash table run from 0 to Tablesize – 1.
- Each key is mapped into some number in the range 0 to Tablesize – 1.
- This mapping is called Hash function.
- Insertion of the data in the hash table is based on the key value obtained from the hash function.
- Using same hash key value, the data can be retrieved from the hash table by few or more Hash key comparison.
- The **load factor** of a hash table is calculated using the formula:

  (Number of data elements in the hash table) / (Size of the hash table)

Factors affecting Hash Table Design

- Hash function
- Table size.
- Collision handling scheme



Simple Hash table with table size = 10

Hash function:

- It is a function, which distributes the keys evenly among the cells in the Hash Table.
- Using the same hash function we can retrieve data from the hash table.
- Hash function is used to implement hash table.
- **The integer value returned by the hash function is called hash key.**
- If the input keys are integer, the commonly used hash function is

H (key) = key % Tablesize

```
typedef unsigned int index;
index Hash ( const char *key , int Tablesize )
{
   unsigned int Hashval = 0 ;
   while ( * key ! = „ \0 „ )
        Hashval + = * key ++ ;
   return ( Hashval % Tablesize ) ;
}
```

A simple hash function

Types of Hash Functions

1. Division Method
2. Mid Square Method
3. Multiplicative Hash Function
4. Digit Folding

1. Division Method:

- It depends on remainder of division.
- Divisor is Table Size.
- Formula is **(H (key) = key % table size)**

E.g. consider the following data or record or key (36, 18, 72, 43, 6) table size = 8

Assume a table with 8 slots:

Hash key = key % table size

4 = 36 % 8

2 = 18 % 8

0 = 72 % 8

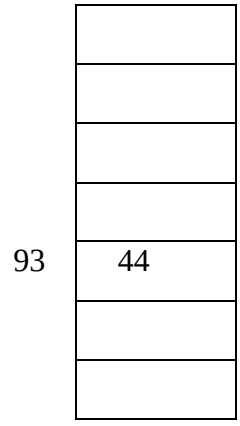
3 = 43 % 8

6 = 6 % 8

[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

2. Mid Square Method:

We first square the item, and then extract some portion of the resulting digits. For example, if the item were 44, we would first compute 44^2=1,936. Extract the middle two digit 93 from the answer. Store the key 44 in the index 93.



3. Multiplicative Hash Function:

Key is multiplied by some constant value.

Hash function is given by,

H(key)=Floor (P * (key * A))

P = Integer constant [e.g. P=50]

A = Constant real number [A=0.61803398987],suggested by Donald Knuth to use this constant

E.g. Key 107

H(107)=Floor(50*(107*0.61803398987))

=Floor(3306.481845)

H(107)=3306

Consider table size is 5000

3306	107
4999	

4. Digit Folding Method:

The folding method for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash key value. For example, if our item was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 (43, 65, 55, 46, 01). After the addition, 43+65+55+46+01, we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case 210 % 11 is 1, so the phone number 436-555-4601 hashes to slot 1.

6-555-4601

Collision:

If two more keys hashes to the same index, the corresponding records cannot be stored in the same location. This condition is known as collision.

Characteristics of Good Hashing Function:

- It should be Simple to compute.
- Number of Collision should be less while placing record in Hash Table.
- **Hash function with no collision ➔ Perfect hash function.**
- Hash Function should produce keys which are distributed uniformly in hash table.
- The hash function should depend upon every bit of the key. Thus the hash function that simply extracts the portion of a key is notsuitable.

Collision Resolution Strategies / Techniques (CRT):

If collision occurs, it should be handled or overcome by applying some technique. Such technique is called CRT.

There are a number of collision resolution techniques, but the most popular are:

- **Separate chaining (**Open Hashing**)**
- **Open addressing**. (Closed Hashing)

Linear Probing

Quadratic Probing

Double Hashing

Separate chaining (Open Hashing)

Open hashing technique.

Implemented using singly linked list concept.

Pointer (ptr) field is added to each record.

When collision occurs, a separate chaining is maintained for colliding data.

- Element inserted in front of the list.

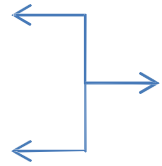
H (key) =key % table size

Two operations are there:-

- Insert
- Find

Structure Definition for Node

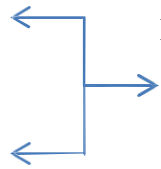
```
typedef Struct node *Position;
Struct node
{
        int data;
        Position  next;
};
```



defines the nodes

Structure Definition for Hash Table

```
typedef Position List;
struct Hashtbl
{
        int Tablesize;
        List * theLists;
};
```



Defines the hash table which contains array of linked list

Initialization for Hash Table for Separate Chaining

```
Hashtable initialize(int Tablesize)
{
        HashTable H;
        int i;
        H = malloc (sizeof(struct HashTbl));                    ➔Allocates table
        H → Tablesize = NextPrime(Tablesize);
        H→the Lists=malloc(sizeof(List) * H→Tablesize);         ➔ Allocates array of list
      for( i = 0; i < H → Tablesize; i++ )
      {
       H → TheLists[i] = malloc(Sizeof(Struct node));     ➔ Allocates list headers
       H → TheLists[i] → next = NULL;
      }
      return H;
}
```

Insert Routine for Separate Chaining

```
void insert (int Key, Hashtable H)
{
  Position P, newnode;                    *[Inserts element in the Front of the list always]*
  List L;
```

```
P = find ( key, H );
if(P = = NULL)
 {
        newnode = malloc(sizeof(Struct node));
        L = H → TheLists[Hash(key,Tablesize)];
        newnode → nex t= L → next;
        newnode → data = key;
       L → next = newnode;
 }}
Position find( int key, Hashtable H){
  Position P, List L;
  L = H →TheLists[Hash(key,Tablesize)];
  P = L → next;
  while(P != NULL && P → data != key)
        P = P → next;
  return P;}
```

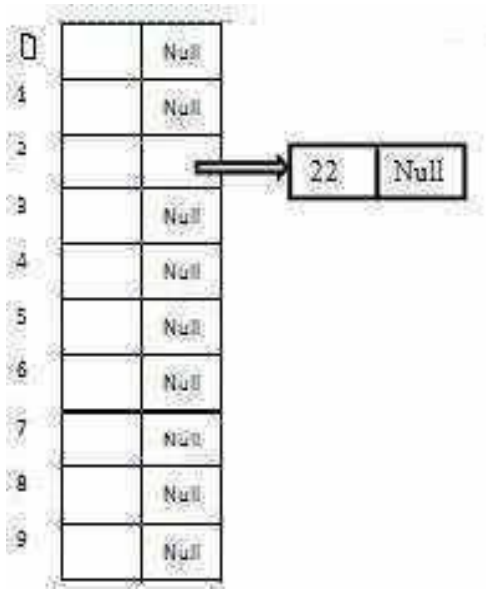
If two keys map to same value, the elements are chained together.

Initial configuration of the hash table with separate chaining. Here we use SLL(Singly Linked List) concept to chain the elements.

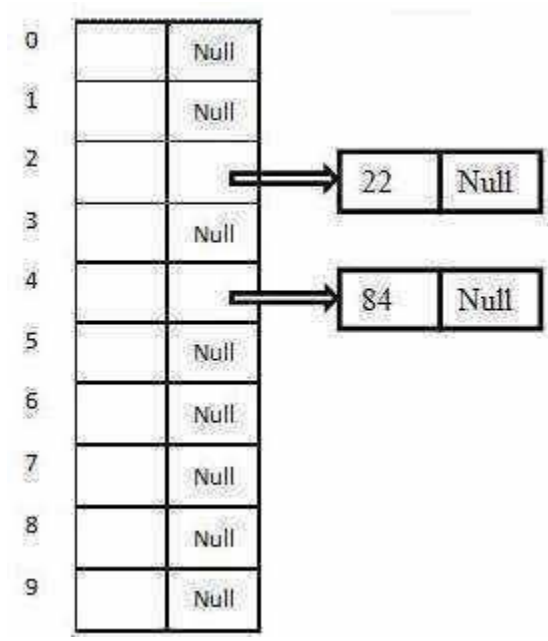
0		NULL
1		NULL
2		NULL
3		NULL
4		NULL
5		NULL
6		NULL
7		NULL
8		NULL
9		NULL

Insert the following four keys 22 84 35 62 into hash table of size 10 using separate chaining. The hash function is
H(key) = key % 10

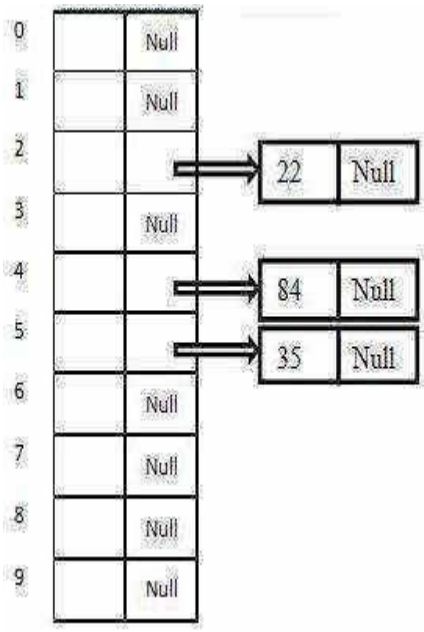
1. **H(22) = 22 % 10 =2**



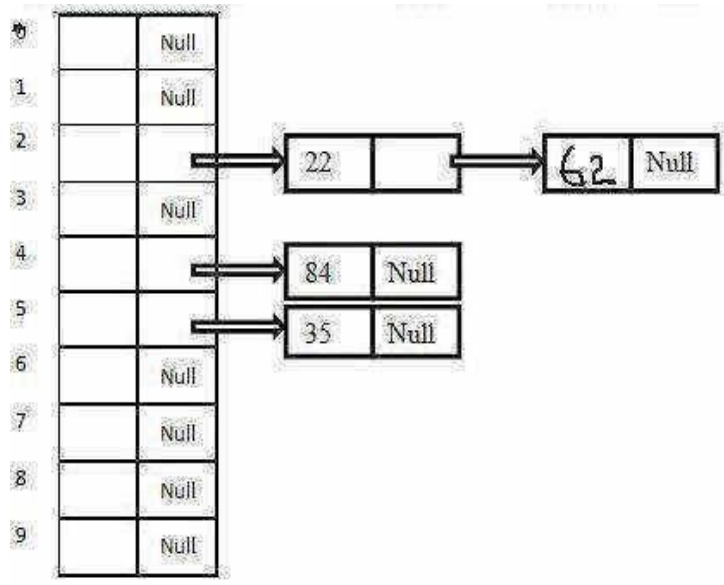
2. 84 % 10 = 4



3.H(35)=35%10=5



4. H(62)=62%10=2



Advantages

1. More number of elements can be inserted using array of Link List

Disadvantages

1. It requires more pointers, which occupies more memory space.
2. Search takes time. Since it takes time to evaluate Hash Function and also to traverse the List

Open Addressing

Closed Hashing
Collision resolution technique
Uses $H_i(X)=(Hash(X)+F(i))$mod Tablesize
When collision occurs, alternative cells are tried until empty cells are found.
Types:-

- Linear Probing
- Quadratic Probing
- Double Hashing

Hash function

- H(key) = key % table size.

Insert Operation

- To insert a key; Use the hash function to identify the list to which the element should be inserted.
- Then traverse the list to check whether the element is already present.
- If exists, increment the count.
- Else the new element is placed at the front of the list.

Linear Probing:

Easiest method to handle collision.
Apply the hash function H (key) = key % table size
$H_i(X)=(Hash(X)+F(i))$mod Tablesize,where F(i)=i.

How to Probing:

- first probe – given a key k, hash to H(key)
- second probe – if H(key)+f(1) is occupied, try H(key)+f(2)
- And so forth.

Probing Properties:

- We force f(0)=0
- The i^{th} probe is to (H (key) +f (i)) %table size.
- If i reach size-1, the probe has failed.
- Depending on f (i), the probe may fail sooner.
- Long sequences of probe are costly.

Probe Sequence is:

- H (key) % table size
- H (key)+1 % Table size
- H (Key)+2 % Table size

1. **H(Key)=Key mod Tablesize**

   This is the common formula that you should apply for any hashing

   If collocation occurs use Formula 2

2. **H(Key)=(H(key)+i) Tablesize**

Where i=1, 2, 3, …… etc

Example: - 89 18 49 58 69; Tablesize=10

1. H(89) =89%10
   =9
2. H(18) =18%10
   =8
3. H(49) =49%10
   =9 ((coloids with 89.So try for next free cell using formula 2))

   i=1 h1(49) = (H(49)+1)%10
   = (9+1)%10
   =10%10
   =0

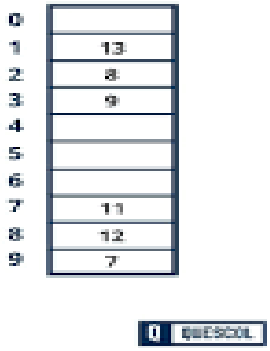
4. H(58) =58%10
   =8 ((colloids with 18))

   i=1 h1(58) = (H(58) +1)%10
   = (8+1) %10
   =9%10
   =9 =>Again collision

   i=2 h2(58) =(H(58)+2)%10
   =(8+2)%10
   =10%10
   =0 =>Again collision

Linear probing

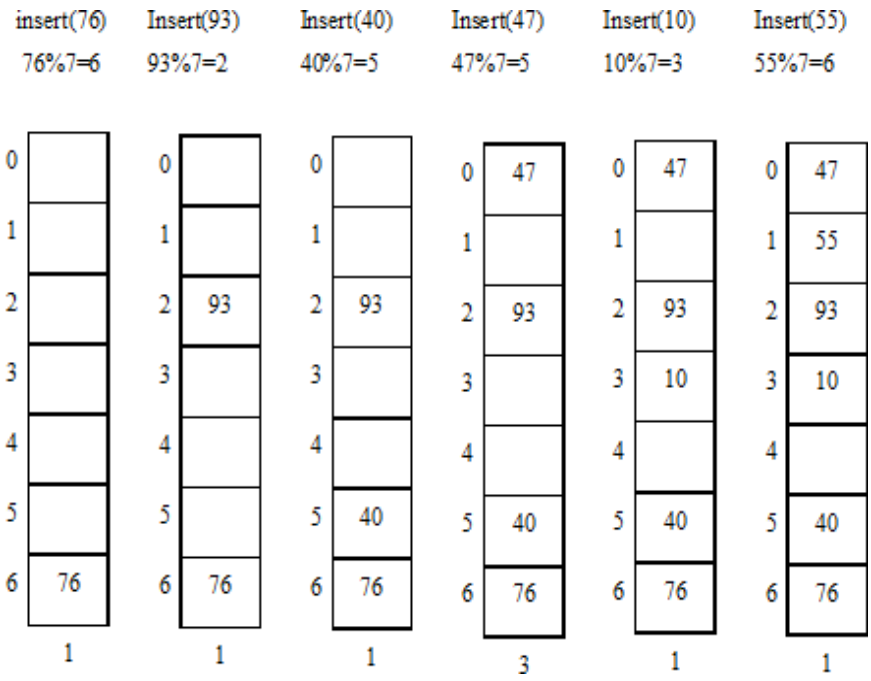
The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at a location generated by h(k), it means collision occurred then we do a sequential search to find the empty location. Here the idea is to place a value in the next available position.



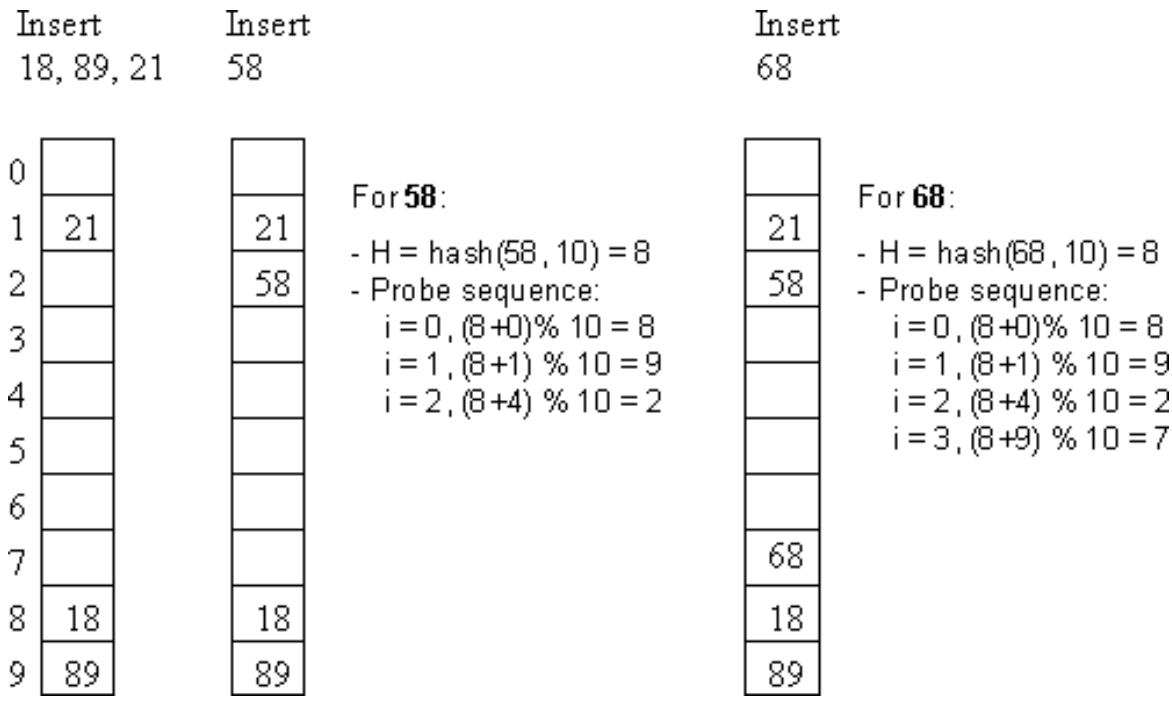
Quadratic Probing

To resolve the primary clustering problem, quadratic probing can be used. With quadratic probing, rather than always moving one spot, move i^2 spots from the point of collision, where i is the number of attempts to resolve the collision.

- Another collision resolution method which distributes items more evenly.



- From the original index H, if the slot is filled, try cells H+12, H+22, H+32,.., H + i2 with wrap-around.
- $H_i(X)=(Hash(X)+F(i))$mod Tablesize,$F(i)=i^2$
- $H_i(X)=(Hash(X)+ i^2)$mod Tablesize



Limitation: at most half of the table can be used as alternative locations to resolve collisions.

This means that once the table is more than half full, it's difficult to find an empty spot. This new problem is known as secondary clustering because elements that hash to the same hash key will always probe the same alternative cells.

Double Hashing

Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions forms the point of collision to insert.

There are a couple of requirements for the second function:

It must never evaluate to 0 must make sure that all cells can be probed.

$H_i(X)=(Hash(X)+i*Hash_2(X))$mod Tablesize

A popular second hash function is:

$Hash_2$ (key) = R - (key % R) where R is a prime number that is smaller than the size of the table.

Table Size = 10 elements
$Hash_1(key) = key \% 10$
$Hash_2(key) = 7 - (k \% 7)$

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 position from [9]

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

Rehashing

Once the hash table gets too full, the running time for operations will start to take too long and may fail. To solve this problem, a table at least twice the size of the original will be built and the elements will be transferred to the new table.

Advantage:

- A programmer doesn‟t worry about table system.
- Simple to implement
- Can be used in other data structure as well

The new size of the hash table:

- should also be prime
- will be used to calculate the new insertion spot (hence the name rehashing)
- This is a very expensive operation! O(N) since there are N elements to rehash and the table size is roughly 2N. This is ok though since it doesn't happen that often.

The question becomes when should the rehashing be applied?

Some possible answers:

- once the table becomes half full
- once an insertion fails

- once a specific load factor has been reached, where load factor is the ratio of the number of elements in the hash table to the table size

Extendible Hashing

- Extendible Hashing is a mechanism for altering the size of the hash table to accommodate new entries when buckets overflow.

Common strategy in internal hashing is to double the hash table and rehash each entry. However, this technique is slow, because writing all pages to disk is too expensive.

Therefore, instead of doubling the whole hash table, we use a directory of pointers to buckets, and double the number of buckets by doubling the directory, splitting just the bucket that overflows.

Since the directory is much smaller than the file, doubling it is much cheaper. Only one page of keys and pointers is split.