SORTING & SEARCHING

What is Sorting

Sorting is a process of ordering a list of elements in either ascending or descending order.

Sorting can be divided into two categories:

1. Internal sorting
2. External sorting

1. Internal sorting

Internal sorting takes place in the main memory of the computer.

Internal sorting can take advantage of the random access nature of the main memory.

2. External sorting

External sorting is carried on secondary storage.

External sorting becomes a necessity if the number of elements to be sorted is too large to fit in main memory.

Sorting Efficiency

In order to find the amount of time required to sort a list of n elements by a particular method, we must find the number of comparisons required to sort.

1. Best case
2. Worst case
3. Average case

Most of the primitive sorting algorithms like bubble sort, selection sort and insertion sort are not suited for sorting a large files.

These algorithms have a timing requirement of $O(n^2)$.

On the contrary, these sorting algorithms require hardly any additional memory space for sorting.

Advance sorting algorithms like quick sort, merge sort and heap sort have a timing requirement of $O(n\ logn)$.

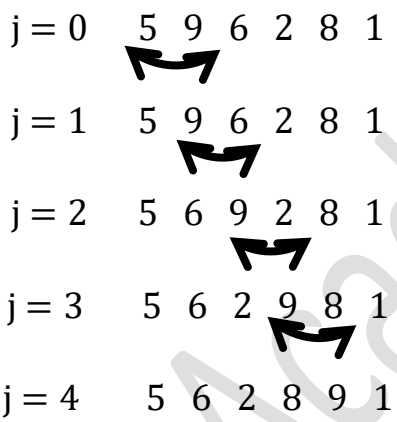
Bubble Sort

Bubble sort is one of the simplest and the most popular sorting method.

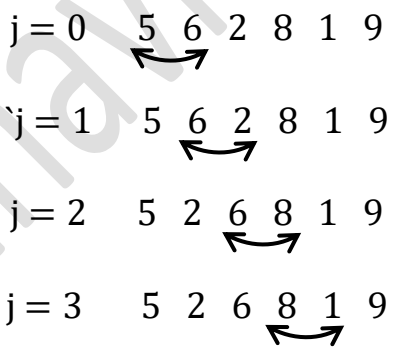
The basic idea behind bubble sort is as a bubble rises up in water, the smallest element goes to beginning.

Original array with n=6 5 9 6 2 8 1

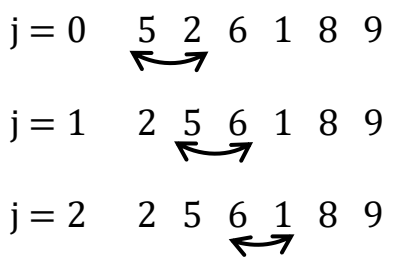
First pass i=1



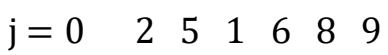
Second pass i=2



Third pass i=3



Fourth pass i=4



j = 1 2 5 1 6 8 9

Fifth pass i= 5

j = 0 2 1 5 6 8 9

Sorted array a[ ] => 1 2 5 6 8 9

Program for sorting an integer array using bubble sort.

```
#include <conio.h>
#include<stdio.h>
void bubble_sort(int[] ,int);
void main()
{
  int a[30] ,n,i;
  printf("\ nEnter no of elements :");
  scanf("% d",&n);
  printf("\ nEnter array elements :");
  for(i=0;i<n;i++)
  {
        scanf ("% d",&a[i] );
  }
  bubble_sort(a,n);
  printf("\ nSorted array is ;");
  for(i=0;i<n;i++)
  {
      printf("% d",a[i] );
  }
```

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

Selection Sort

Selection sort is very simple sorting method.

5	9	1	11	2	4	Original array
1	9	5	11	2	4	After first pass
1	2	5	11	9	4	After second pass
1	2	4	11	9	5	After third pass
1	2	4	5	9	11	After forth pass

1 2 4 5 9 11 After fifth pass

Write a program to sort number in ascending order using selection sort.

```
#include <conio.h>
#include<stdio.h>
void selection_sort(int[ ],int);
void main()
{
   int a[50],n,i;
   printf("\nEnter no of elements :");
   scanf("%d",&n);
   printf("\nEnter array elements :");
  for(i=0;i<n;i++)
   {
     scanf ("%d",&a[i]);
   }
  selection_sort(a,n);
  printf("\n sorted elements are :");
  for(i=0;i<n;i++)
   {
      printf("%d",a[i]);
   }
   getch();
}
```

```
void selection_sort(int a[] ,int n)
{
  int i,j,k,temp;
  for(i=0;i<n-i;i++)
  {
      k=i;
      for(j=i+1;j<n;j++)
      {
          if(a[j] <a[k] )
              {
               k=j;
          }
      }
      if(k!=i)
      {
          temp=a[i] ;
          a[i] =a[k] ;
          a[k] =temp;
      }
}
```

Insertion Sort

An element can always be placed at a right place in sorted list of element

Insertion sort is based on the principle of inserting the element at its correct place in a previously sorted list. I can be varied 1 to n-1 to sort the entire array.

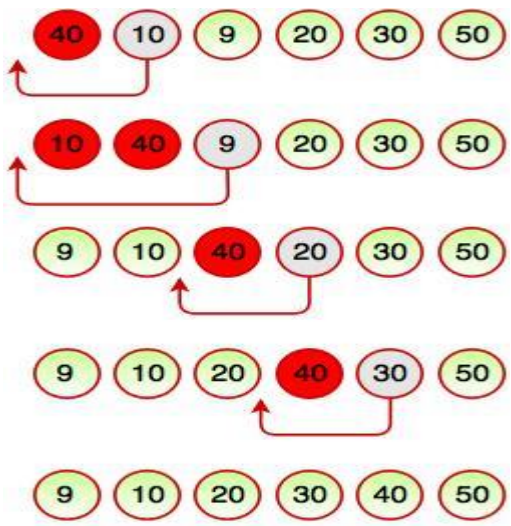


Fig. Working of Insertion Sort

Write a program to sort number in ascending order using insertion sort.

```
#include <conio.h>
#include<stdio.h>
void insertion_sort(int[ ],int);
void main()
{
   int a[50],n,i;
   printf("\nEnter no of elements :");
   scanf("%d",&n);
   printf("\nEnter array elements :");
  for(i=0;i<n;i++)
     scanf ("%d",&a[i]);
  insertion_sort(a,n);
  printf("\n sorted elements are :");
  for(i=0;i<n;i++)
  {
     printf("%d",a[i]);
  }
```

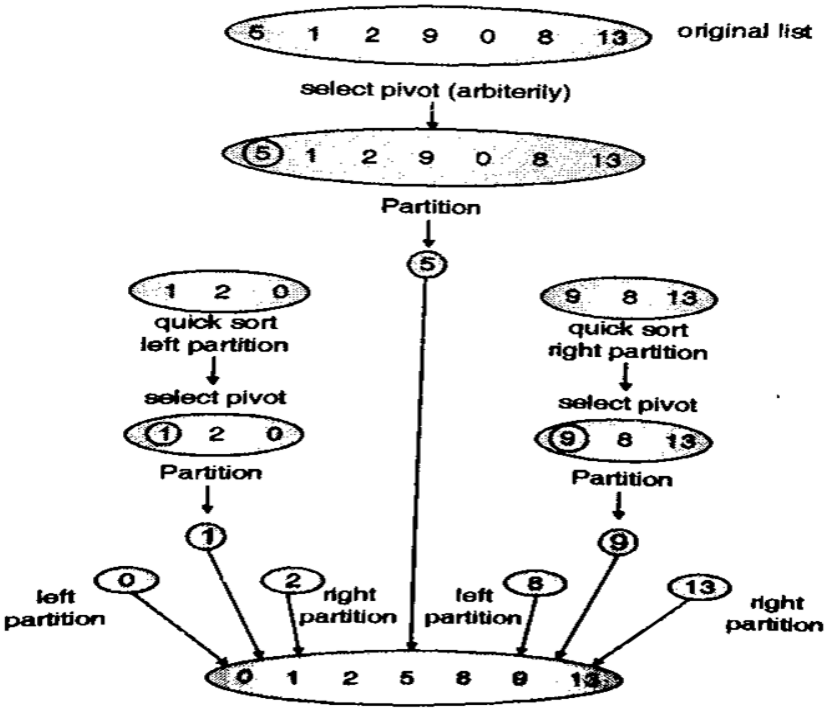
```
  getch();
}
void insertion_sort(int a[] , int n)
{
        int I, j, temp;
        for(i=1;i<n;i++)
        {
                temp=a[i] ;
                for(j=i-1; j>=0 && a[j] >temp; j--)
                {
                        a[j+1] =a[j] ;
                }
                a[j+1] =Temp;
        }
}
```

Quick Sort

Quick sort is the fastes internal sorting algorithm with the time complexity= O(n log n).

The basic algorithm to sort an array a [] of n elements can be described recursively as follows

1. Pick any element V in a[]. This is called pivot.
2. Rearrange elements of the xi < V left of V.
3. If the place of the V after re-arrangement is j, all elements with value less than V, appear in a[0],a[1]…a[j-1] and all those with value grater than V appear in a[j+1]…a[n-1].




```
void Quick_Sort(int a[], int l, int u)
{
        int j;
        if(l<u)
        {
                J=partition(a,l,u);
                Quick_sort(a,l,j-1);
                Quick_sort(a,j+1,u);
        }
}
Int partition (int a[], int l, int u)
{
        int v,l,j,temp;
        V=a[l];
        i=l;
```

```
    j=u+1;
    do
    {
        do
        {
            i++;
        }while(a[i]<v && i<=u);
        do
        {
            j--;
        }while(a[j]>v);
        if(i<j)
        {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }while(i<j);
    a[l]=a[j];
    a[j]=v;
    return (j);
}
```

Write a program to sort number in ascending order using Quick sort.

```
#include <conio.h>
#include<stdio.h>
```

```
void Quick_sort(int[ ] ,int);
void main()
{
   int a[50] ,n, i;
   printf("\ nEnter no of elements :");
   scanf("% d",&n);
   printf("\ nEnter array elements :");
  for(i=0;i<n;i++)
     scanf ("% d",&a[i] );
  Quick_sort(a,0,n);
  printf("\ n sorted elements are :");
  for(i=0;i<n;i++)
   {
     printf("% d",a[i] );
   }
   getch();
}
```

Merge sort

Merge sort runs in O(N Log N) running time. It is a very efficient sorting algorithm with near optimal number of comparisons

Basic operation in merge sort is that of merging of two sorted list into one sorted list. Merging operation has a liner time complex

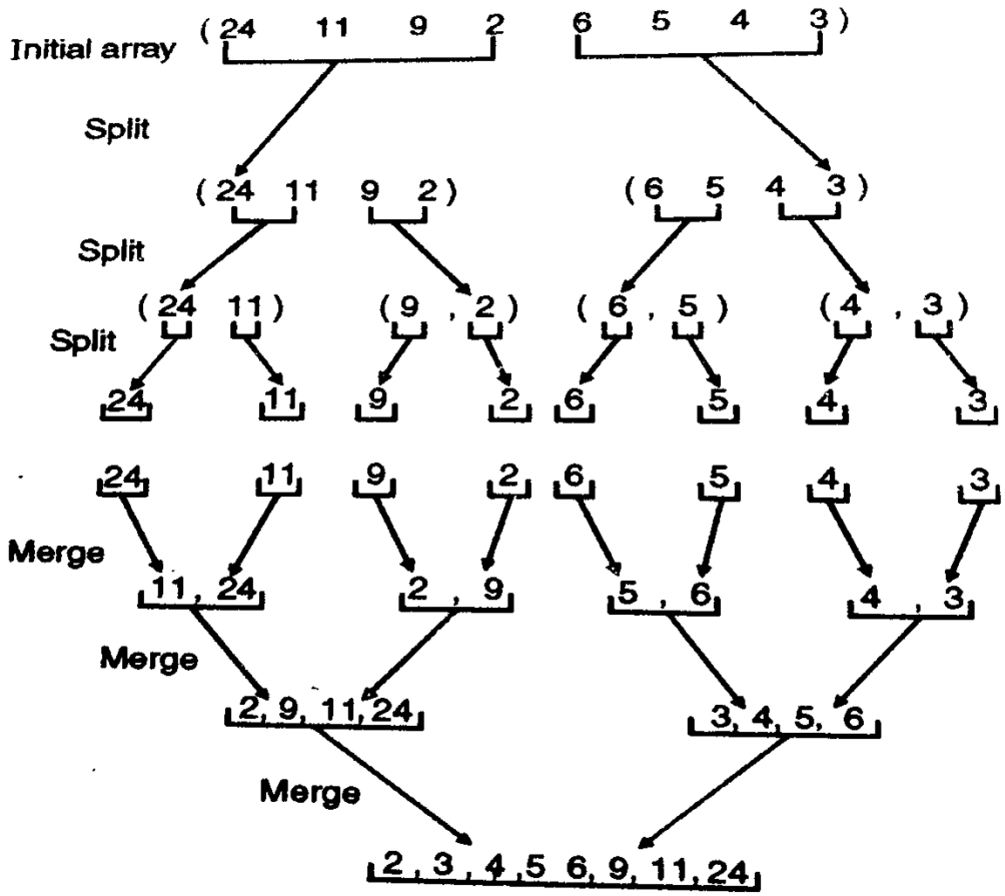
Advantages and Disadvantages of Merge sort:

Advantages

1. It has a timing complexity of O(n log n).
2. It can be used both for internal as well as external sorting.
3. It is a stable sorting algorithm.

Disadvantages

1. It requires an additional memory for storing of merged data during merging phase.
2. It can take advantage of partially sorted nature of data. Number of passes are fixed.




```
Void merge_sort(int a[] , int I, int j)
{
    Int k;
    If(i<j)
    {
        k=(i+j)/2;
        Merge_sort(a,i,k);
        Merge_sort(a,k+1,j);
```

```
            Merge(a,i,k,j);
        }
}
Void merge( int a[] , int l, int m, int u)
{
        Int c[20] ;
        int i,j,k;
        i=l;
        j=m+1;
        k=0;
        while(i<=m && j<=u)
        {
            If(a[i] <a[j] )
            {
                c[k] =a[i] ;
                k++; i++;
            }
            else
            {
                c[k] =a[j] ;
                k++; j++;
            }
        }
        while (i<=m)
        {
            c[k] =a[i] ;
```

```
            k++; i++;
        }
        while(j<=u)
        {
            c[k] =a[j] ;
            k++; j++;
        }
        for(i=l,j=0;i<=u;i++,j++)
        {
            a[i] =c[j] ;
        }
}
```

Radix sort

Radix sort is generalization of bucket sort. To sort decimal numbers, where the redix or base is 10, we need 10 buckets . These buckets are numbered 0,1,2,3,4,5,6,7,8,9. Sorting is done in passes

Number of passes required to sort using shell sort is equal to number of digits in the largest number in the list

Range	Passes
0to 99	2 passes
0to999	3 passes
0to9999	4 passes

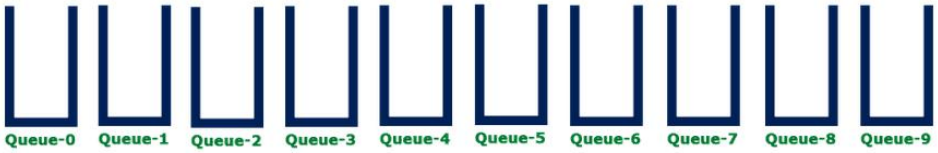
In the first pass, number are sorted on least significant digit. Numbers with the same least significant digit are sorted in the same bucket

- In the 2nd pass, numbers are sorted on the second least significant digit.
- At the end of every pass, numbers in buckets are merged to produce a common list.
- Number of passes depends on the range of numbers being sorted.
- The following example shows the action of radix sort.

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.



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

8<u>2</u>, 90<u>1</u>, 10<u>0</u>, 1<u>2</u>, 15<u>0</u>, 7<u>7</u>, 5<u>5</u> & 2<u>3</u>

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

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.

Heap sort

Heap sort is one of the sorting algorithms used to arrange a list of elements in order.

Heapsort algorithm uses one of the tree concepts called **Heap Tree**.

In this sorting algorithm, we use **Max Heap** to arrange list of elements in Descending order and **Min Heap** to arrange list elements in Ascending order.

Step by Step Process

The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

Step 1 - Construct a **Binary Tree** with given list of Elements.

Step 2 - Transform the Binary Tree into **Min Heap.**

Step 3 - Delete the root element from Min Heap using **Heapify** method.

Step 4 - Put the deleted element into the Sorted list.

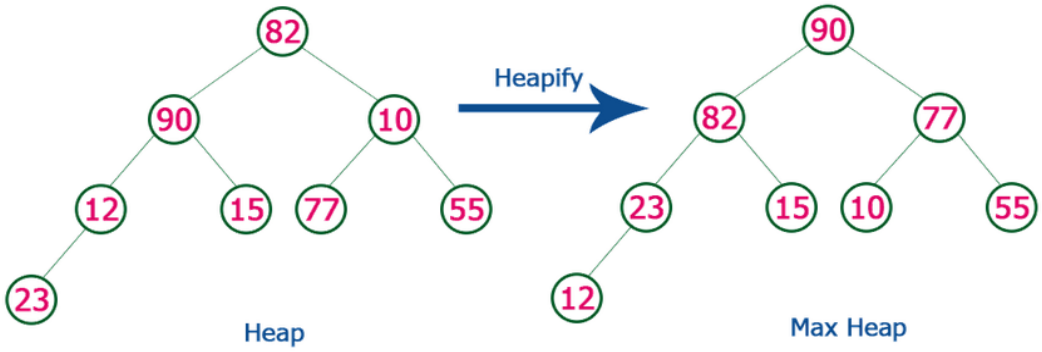
Step 5 - Repeat the same until Min Heap becomes empty.

Step 6 - Display the sorted list.

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

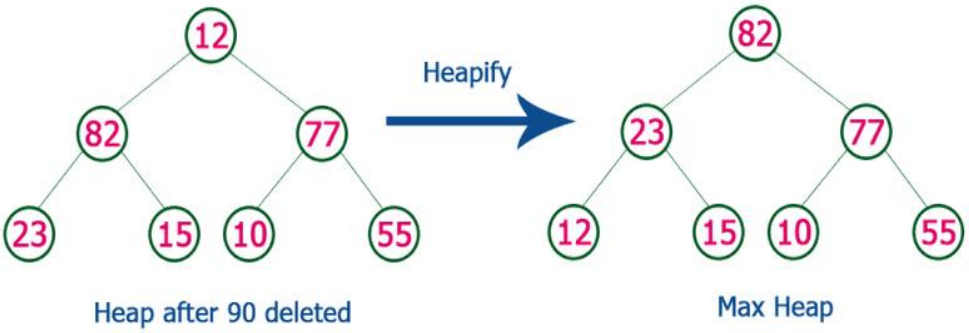
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

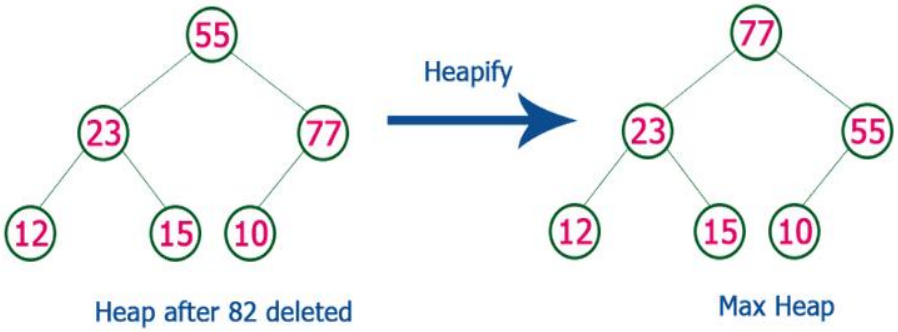
Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, 90

Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 82 with 55.

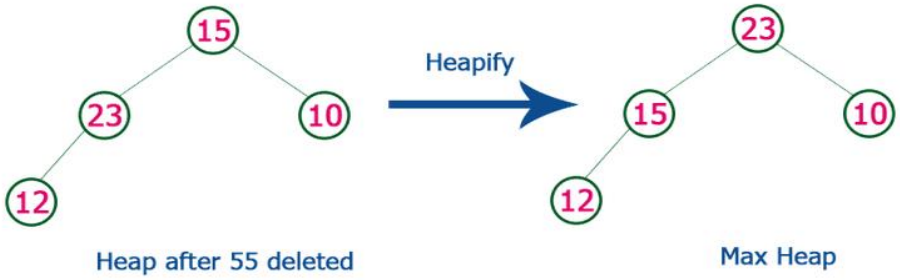
12, 55, 77, 23, 15, 10, 82, 90

Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, 77, 82, 90

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

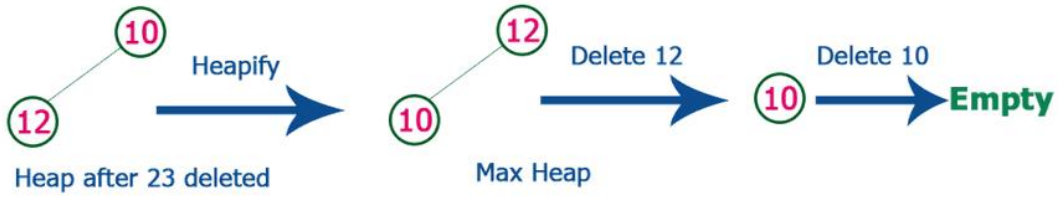
12, 15, 10, 23, 55, 77, 82, 90

Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 23 with 12.

12, 15, 10, 23, 55, 77, 82, 90

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Searching

Searching is a technique of finding an element in a given list of elements

Searching technique should be able to locate the element to be searched as quickly as possible.

The complexity of any searching algorithm depends on number of comparisons required to find the element. Performance of searching algorithm can be found by counting the number of comparisons in order to find the given element.

Sequential search (linear search):

In sequential search elements are examined sequentially starting from the first element.

The process of searching terminates when the list is exhausted or a comparison result in success.

Algorithm for searching an element 'key' in an array 'a[]' having elements:

```
Int sequential( int a[], int key, int n)
{
    int i=0'
    While (i<n)
    {
```

```
            if(a[i]==key)
            {
                  return (i);
            }
            i++;
      }
return (-1);
}
```

Write a program for sequential search

```
#include <conio.h>
#include<stdio.h>
void sequential_search(int[ ] ,int);
void main()
{
   int a[50] ,n, i;
   printf("\ nEnter no of elements :");
   scanf("%d",&n);
   printf("\ nEnter array elements :");
  for(i=0;i<n;i++)
     scanf ("%d",&a[i] );
printf("Element to be search \ n");
scanf("%d", &key);
if(i==-1)
{
```

```
        Printf("Not found \n");
}
Else
{
        Printf("Found at location %d", i+1);
}
  getch();
}
```

Binary Search

Linear search is not applicable when data is on large volume,

Binary search exhibits much better timing behavior in case of large volume of data.

Algorithm

Binary search is implemented using following steps...

- Step 1 - Read the search element from the user.
- Step 2 - Find the middle element in the sorted list.
- 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.
- 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.
- 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.
- 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.

```
int bin_Search( int a[], int key, int n)
{
    int i,j, c;
    i=0;
    j=n;
    c=(i+j)/2;
    While (key==a[c])
    {
        If(key <a[c])
        {
            j=c-1;
        }
        else
        {
            i=c+1;
        }
        c=(i+j)/2;
    }
    If(i<=j)
        Return c;
```

```
    Return (-1);
    return (i);
}
i++;
}
return (-1);
}
```

What is Hashing

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values. However, in cases where the keys are large and cannot be used directly as an index, you should use *hashing*.

In hashing, large keys are converted into small keys by using **hash functions**.

The values are then stored in a data structure called **hash table**.

The idea of hashing is to distribute entries (key/value pairs) uniformly across an array.

Each element is assigned a key (converted key). By using that key you can access the element in **O(1)** time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hashing is implemented in two steps:

1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.

2. The element is stored in the hash table where it can be quickly retrieved using hashed key.

   hash = hashfunc(key)
   index = hash % array_size

In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).

- Suppose we have integer items {26, 70, 18, 31, 54, 93}. One common method of determining a hash key is the division method of hashing and the formula is :

Hash Key = Key Value % Number of Slots in the Table

- Division method or reminder method takes an item and divides it by the table size and returns the remainder as its hash value.

Data Item	Value % No. of Slots	Hash Value
26	26 % 10 = 6	6
70	70 % 10 = 0	0
18	18 % 10 = 8	8
31	31 % 10 = 1	1
54	54 % 10 = 4	4
93	93 % 10 = 3	3

Hash function

To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:

1. Easy to compute: It should be easy to compute and must not become an algorithm in itself.
2. Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
3. Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

   Note: Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

Need for a good hash function

The hash function will compute the same index for all the strings and the strings will be stored in the hash table in the following format. As the index of all the strings is the same, you can create a list on that index and insert all the strings in that list.

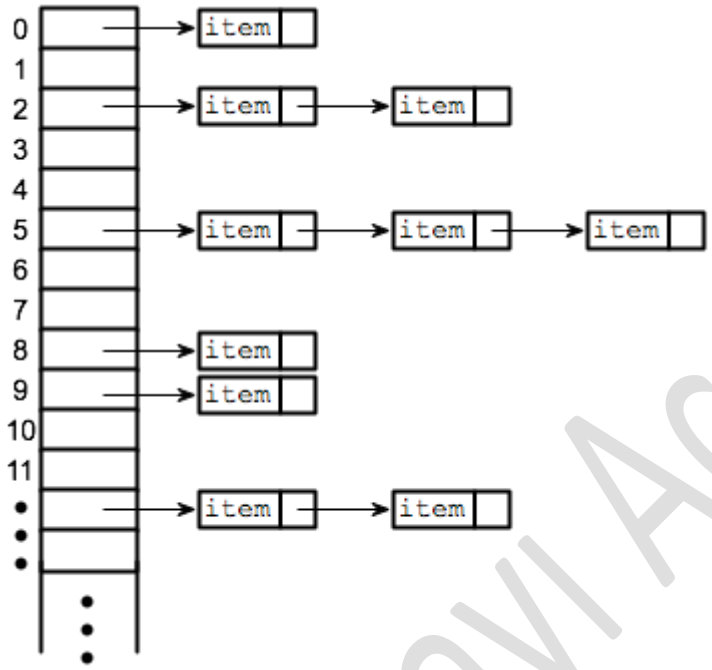
Hash table

A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched. By using a good hash function, hashing can work well. Under reasonable assumptions, the average time required to search for an element in a hash table is **O(1)**.

Collision resolution techniques

Separate chaining (open hashing)

Separate chaining is one of the most commonly used collision resolution techniques. It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list. If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.



The cost of a lookup is that of scanning the entries of the selected linked list for the required key. If the distribution of the keys is sufficiently uniform, then the average cost of a lookup depends only on the average number of keys per linked list. For this reason, chained hash tables remain effective even when the number of table entries (N) is much higher than the number of slots.

For separate chaining, the worst-case scenario is when all the entries are inserted into the same linked list. The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number (N) of entries in the table.

Linear probing (open addressing or closed hashing)

In open addressing, instead of in linked lists, all entry records are stored in the array itself. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.

The probe sequence is the sequence that is followed while traversing through entries. In different probe sequences, you can have different intervals between successive entry slots or probes.

When searching for an entry, the array is scanned in the same sequence until either the target element is found or an unused slot is found. This indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value.

Linear probing is when the interval between successive probes is fixed (usually to 1). Let's assume that the hashed index for a particular entry is **index**. The probing sequence for linear probing will be:

index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 2) % hashTableSize
index = (index + 3) % hashTableSize

Quadratic Probing

Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots. Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot. The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.

Let us assume that the hashed index for an entry is **index** and at **index** there is an occupied slot. The probe sequence will be as follows:

index = index % hashTableSize
index = (index + 1^2) % hashTableSize
index = (index + 2^2) % hashTableSize
index = (index + 3^2) % hashTableSize

Example 10.18.2 : Using linear probing and quadratic probing insert the following values in a hash table of size 10. Show how many collisions occur in each technique : 99, 33, 23, 44, 56, 43, 19 **MU - Dec. 2013, May 2014**

Solution :

1. Linear probing :

	Empty table	After 99	After 33	After 23	After 44	After 56	After 43	After 19
0								19*
1								
2								
3			33	33	33	33	33	33
4				23*	23*	23*	23*	23*
5					44*	44*	44*	44*
6						56	56	56
7							43*	43*
8								
9		99	99	99	99	99	99	99

* = Collision

No. of collisions = 4