Sorting

Sorting is nothing but storage of data in sorted order; it can be in ascending or descending order. The term Sorting comes into picture with the term Searching. There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.

Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted

Many different algorithms for sorting

- Bubble Sort
- Shell Sort
- Radix Sort
- Insertion Sort
- Heap Sort
- Quick Sort
- Merge Sort

Bubble Sort

Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, you mustn't swap the element. Then, again second and third elements are compared and swapped if it is necessary and this process go on until last and second last element is compared and swapped. This completes the first step of bubble sort.

If there are *n* elements to be sorted then, the process mentioned above should be repeated n-1 times to get required result. But, for better performance, in second step, last and second last elements are not compared because, the proper element is automatically placed at last after first step. Similarly, in third step, last and second last and second last and third last elements are not compared and so on.

1. Start
2. Input n
3. Input n numbers
4. for i=0 to n-2 do
   for j=0 to n-i-1 do
   if(a[j]>a[j+1]) then
   swap a[j] & a[j+1]
5. Display All elements
6. Stop

Example
23 2 56 9 8 10

Iteration1
2 23 56 9 8 10
2 23 56 9 8 10

2 23 9 56 8 10
2 23 9 8 56 10
2 23 9 8 10 **56**

Iteration2
2 23 9 8 10 56
2 9 23 8 10 56
2 9 8 23 10 56
2 9 8 10 **23 56**

Iteration3
2 9 8 10 23 56
2 8 9 10 23 56
2 8 9 **10 23 56**

Iteration4
2 8 9 10 23 56
2 8 **9 10 23 56**

Iteration5
2 8 9 10 23 56

C Program

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

void main()
{
        int a[10],n, i, j,k, temp;
        clrscr();
        printf("Enter how many numbers you want to sort");
        scanf("%d",&n);
        printf("Enter element\n");
        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("Sorted elements are\n");
        for(i=0;i<n;i++)
                printf("%d ",a[i]);
        getch();
}
```

Modified Bubble sort

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

void main()
{
        int a[10],n, i, j, temp,flag=1,k;
        clrscr();
        printf("Enter how many numbers you want to sort");
        scanf("%d",&n);
        printf("Enter element\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);

        for(i=0; i<n-1 &&flag==1; i++)
        {
          flag=0;
          for(j=0; j<n-i-1; j++)
          {
                if( a[j] > a[j+1])
                {
                    flag=1;
                    temp = a[j];
                    a[j] = a[j+1];
                    a[j+1] = temp;
                }
          }
        }
        printf("Sorted elements are\n");
        for(i=0;i<n;i++)
                printf("%d ",a[i]);
        getch();
}
```

Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one element at a time. In this sort elements are divided into two parts sorted element and unsorted elements. Consider first element as a sorted part and remaining (n-1) elements present in unsorted part.

Take a first element from unsorted part and insert that element into proper position in sorted part for that you may require to shift element (greater elements than selected element) at right. Repeat these steps until unsorted sort contain 0 element. At the end you get sorted array.

Example
23 2 56 9 8 10

Index	0	1	2	3	4	5
Element	23	2	56	9	8	10

Index	0
Element	23

Sorted

Index	1	2	3	4	5
Element	2	56	9	8	10

Unsorted

Index	0	1
Element	2	23

Sorted

Index	2	3	4	5
Element	56	9	8	10

Unsorted

Index	0	1	2
Element	2	23	56

Sorted

Index	3	4	5
Element	9	8	10

Unsorted

Index	0	1	2	3
Element	2	9	23	56

Sorted

Index	4	5
Element	8	10

Unsorted

Index	0	1	2	3	4
Element	2	8	9	23	56

Sorted

Index	5
Element	10

Unsorted

Index	0	1	2	3	4	5
Element	2	8	9	10	23	56

Sorted

Algorithm

1. Start
2. Input n
3. Input n numbers
4. for i=1 to n-1 do
   temp=a[i]
   j=i-1
   while(temp<a[j] && j>=0)
   {
   a[j+1]=a[j];
   j--;
   }
   A[j+1]=temp
5. Display All elements
6. Stop

C Program

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

void insertion(int[], int);
void main()
{
  int i, n, a[20];
  clrscr();

  printf("Enter total elements: ");
  scanf("%d", &n);

  printf("Enter %d elements: ", n);
  for (i = 0; i < n; i++)
  {
    scanf("%d", &a[i]);
  }
  insertion(a,n);
  printf("After Sorting: ");
  for (i = 0; i < n; i++)
  {
    printf("%d ", a[i]);
  }
  getch();
}

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

```
    temp = a[i];
    j = i - 1;
    while ((temp < a[j]) && (j >= 0))
    {
        a[j + 1] = a[j];
        j = j - 1;
    }
    a[j + 1] = temp;
  }
}
```

Quick sort:-

Quick sort is work on divide & conquer method in this method the element to be sorted and partitioning into three segment left segment, right segment & Middle segment

The element present in left segment is less then middle segment & the element present in the right segment is greater than middle segment

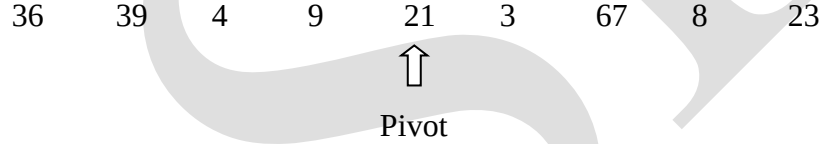
The middle segment contains exactly one element called **pivot** element.

The quick sort algorithm work as follow:-

1) Select an element pivot from the array element
2) Rearrange the element in the array in such a way that all the element that are less then pivot element appears before pivot element & greater then appears after the pivot element
3) After search partitioning the pivot element is place in its final position. This is called partition operation
4) Recursively sort the left segment & right segment using above step.

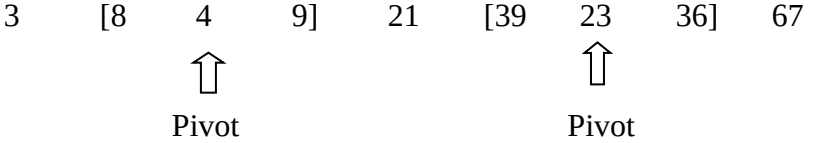
Example:-

Step 1:-



Step 2:-

Step 3:-



Step 4:-

3 4 [8 9] 21 23 [39 36] 67

Pivot

Pivot

Step 5:-

3 4 8 [9] 21 23 [36] 39 67

Step 6:-

3 4 8 9 21 23 36 39 67

Program:-

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

void quicksort(int list[],int lo,int hi);

void main()
{
	int list[10],i,n,lo,hi;
	clrscr();
	printf("\nEnter No. of Elements in Array: ");
	scanf("%d",&n);

	lo=0;
	hi=n-1;

	printf("\nEnter the elements of array: ");
	for(i=0;i<n;i++)
	{
		scanf("%d",&list[i]);
	}

	printf("\nElements before sorting: ");
	for(i=0;i<n;i++)
	{
		printf("%d ",list[i]);
	}

	quicksort(list,lo,hi);

	printf("\nElements after sorting: ");
	for(i=0;i<n;i++)
	{
```

```
                printf("%d ",list[i]);
        }

        getch();
}

void quicksort(int list[],int lo,int hi)
{
        int i,j,x,temp;
        i=lo;j=hi;
        x=list[(lo+hi)/2];

        do
        {
                while(list[i]<x)
                        i++;
                while(list[j]>x)
                        j--;
                if(i<=j)
                {
                        temp=list[i];
                        list[i]=list[j];
                        list[j]=temp;
                        i++;
                        j--;
                }
        }while(i<=j);

        if(lo<j)
                quicksort(list,lo,j);

        if(i<hi)
                quicksort(list,i,hi);
}
```

Merge Sort

Merge Sort is a sorting algorithm that uses the divide, conquer and combine algorithmic method. In this method division is dynamically carried out.

1. **Divide**:- Divide means partitioning elements of array into two sub array of n/2 elements in each sub array.

   If any sub array contains 0 or 1 element then it is already sorted.
2. **Conquer:** - sorting two sub arrays recursively using merge sort.
3. **Combine**: - Merge two sorted sub arrays of size n/2 into 1 array of n elements.

   It will focus on into performance.

Step 1. 36 39 4 9 21 3 67 8 23
/ \
Step 2. [36 39 4 9 21] [3 67 8 23]
/ \ / \
Step 3. [36 39 4] [9 21] [3 67] [8 23]
/ \ / \ / \ / \
Step 4. [36 39] [4] [9] [21] [3] [67] [8] [23]
/ \
Step 5. [36] [39] [4] [9] [21] [3] [67] [8] [23]
\ /
Step 6. [36 39] [4] [9] [21] [3] [67] [8] [23]
\ / \ / \ / \ /
Step 7. [4 36 39] [9 21] [3 67] [8 23]
\ / \ /
Step 8. [4 9 21 36 39] [3 823 67]
\ /
Step 9. [3 4 8 9 21 23 36 39 67]

Algorithm:

```
Algorithm  sort (a, l, h)
{
  int mid;
  if (l == h)
       return;
  else
      if (l < h)
      {
   mid = (l+h)/2;
   sort (a, l, mid);
   sort (a, mid +1, h);
   merge (a, l, mid, h);
      }
 }
 Algorithm merge (a, low,  mid , high)
 {
  int h, i, j, k;
  int b[10];
  h = low ;
  i = low;
  j = mid +1;
  while ((h <= mid) && (j <= high))
  {
    if (a[h] <= a[j])
    {
     b[i] = a[h];
     h ++;
```

```
    }
    else
    {
     b[i] = a[j];
     j ++;
    }
    i++;
   }
   if (h > mid)
    for (k =j; k <= high; k++)
    {
     b[i] = a[k];
     i++;
    }    else
    for (k =h; k <= mid; k++)
    {
b[i] = a[k];
                    i ++;
}
    for (k =low ; k <= high; k++)
    a[k] = b [k];
   }
```

C Program

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

void mergesort(int[], int, int, int);
void sort(int[],int, int);
void main()
{
        int list[10],i,n,lo,hi;
        clrscr();
        printf("\nEnter No. of Elements in Array: ");
        scanf("%d",&n);

        lo=0;
        hi=n-1;

        printf("\nEnter the elements of array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d",&list[i]);
        }

        sort(list,lo,hi);

        printf("\nElements after sorting: ");
```

```
        for(i=0;i<n;i++)
        {
                printf("%d ",list[i]);
        }

        getch();
}

void sort (int a[],int l, int h)
{
  int mid;
  if (l == h)
        return;
  else
    if (l < h)
    {
        mid = (l+h)/2;
        sort (a, l, mid);
        sort (a, mid +1, h);
        mergesort (a, l, mid, h);
    }
}
void mergesort (int a[],int low,int  mid ,int high)
{
  int h, i, j, k;
  int b[10];
  h = low ;
  i = low;
  j = mid +1;
  while ((h <= mid) && (j <= high))
  {
    if (a[h] <= a[j])
    {
     b[i] = a[h];
     h ++;
    }
    else
    {
     b[i] = a[j];
     j ++;
    }
    i++;
   }
   if (h > mid)
    for (k =j; k <= high; k++)
    {
     b[i] = a[k];
     i++;
    }    else
```

```
for (k =h; k <= mid; k++)
{      b[i] = a[k];
 i ++;     }
for (k =low ; k <= high; k++)
a[k] = b [k];
}
```

Heap Sort

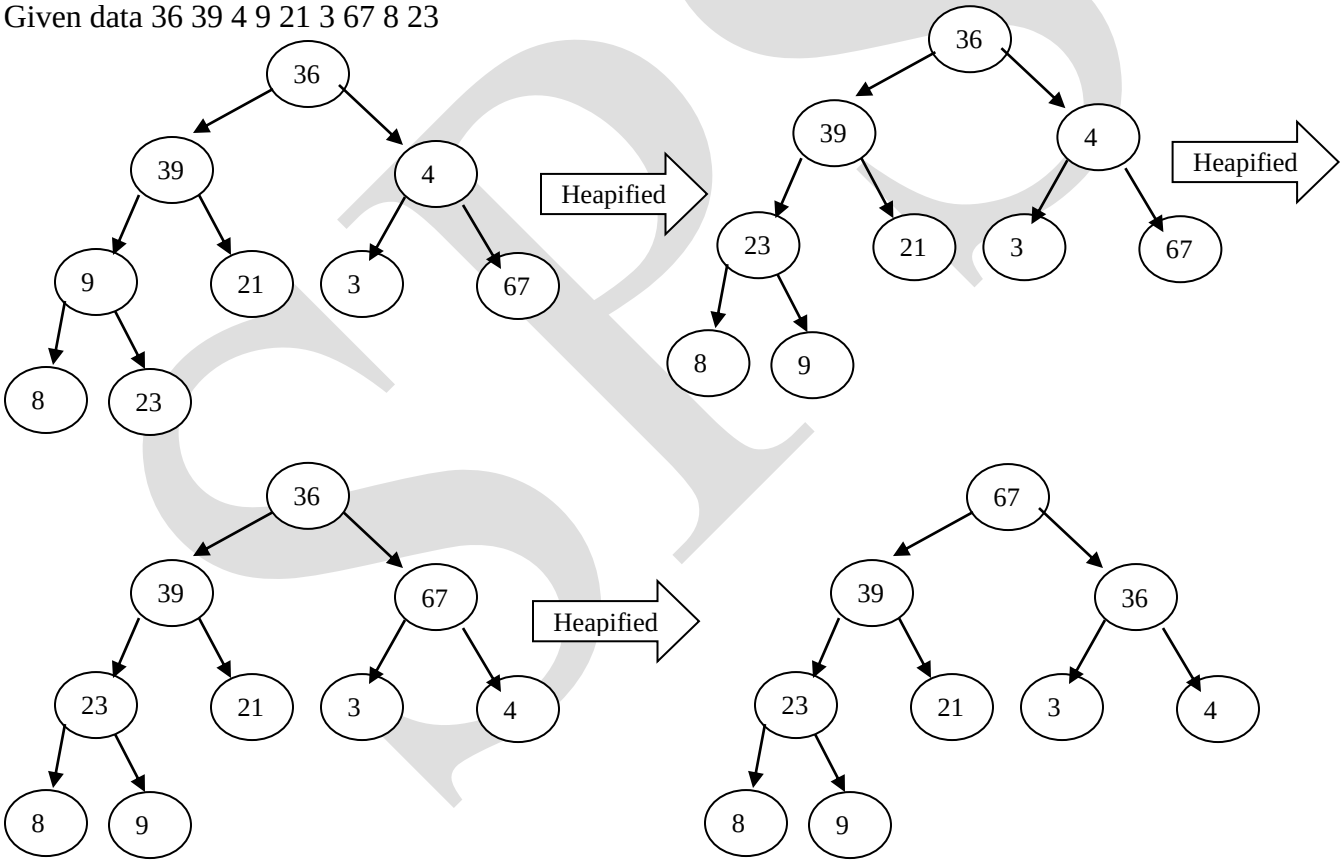
Heap sort is a sorting method. It works in two stages.

1. Construct heap tree
2. Repeatedly delete root element from heap tree.

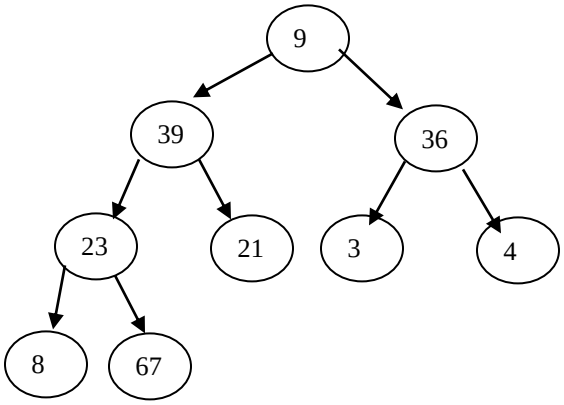
In heap sort first construct heap tree & then delete the root node from heap tree & put that node in the last position of the array. Then if necessary then heapify heap tree. Then again delete the new root element & put it in second last position of array. Repeat this process for all element & you get sorted array.

Example

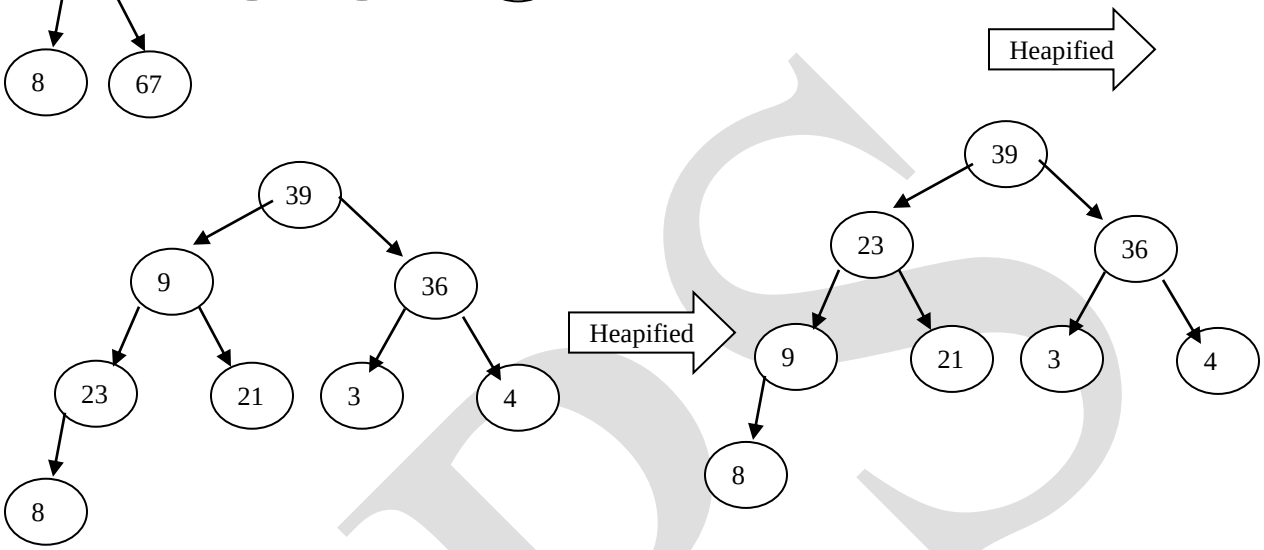
Given data 36 39 4 9 21 3 67 8 23



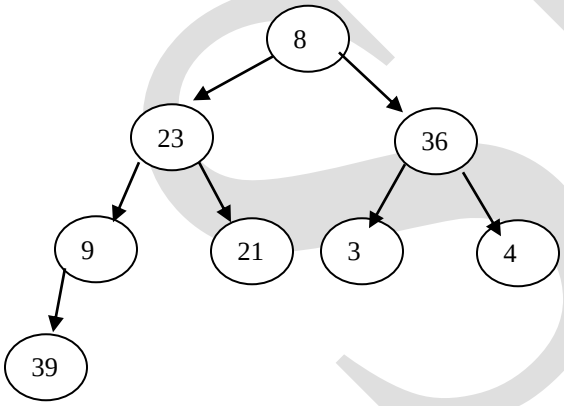
Delete root node i.e. 67. Swap root element and last element of tree.



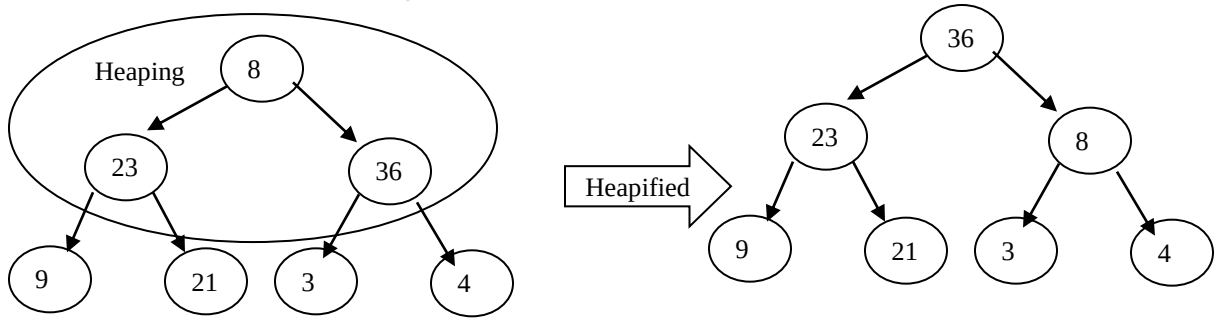
9	39	36	23	21	3	4	8	**67**



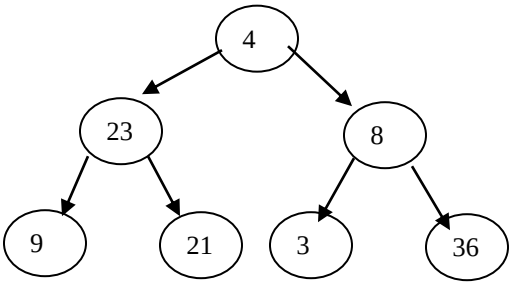
Delete root node i.e. 39. Swap root element and last element of tree.



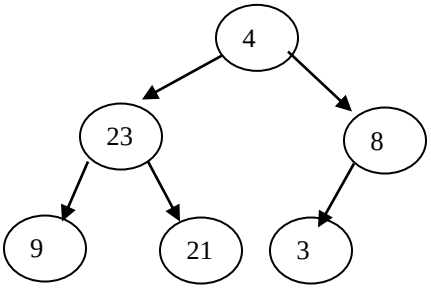
8	23	36	9	21	3	4	**39**	**67**

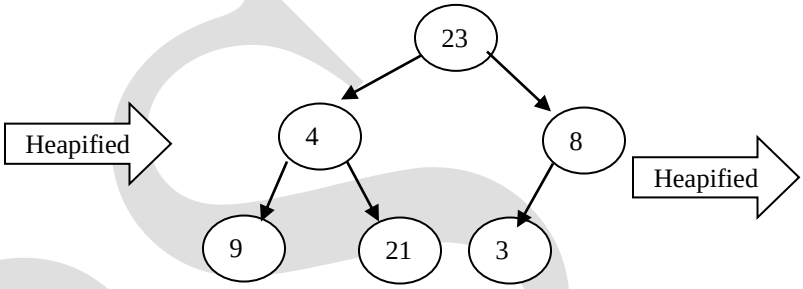


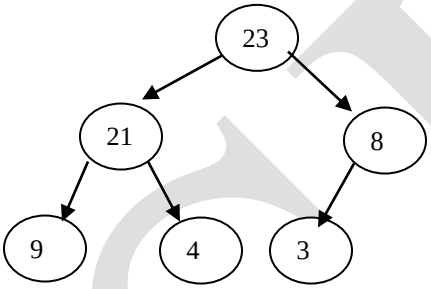
Delete root node i.e. 36. Swap root element and last element of tree.



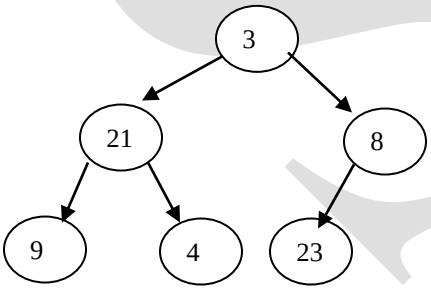
4	23	8	9	21	3	**36**	**39**	**67**



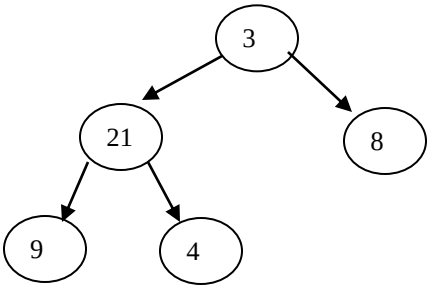


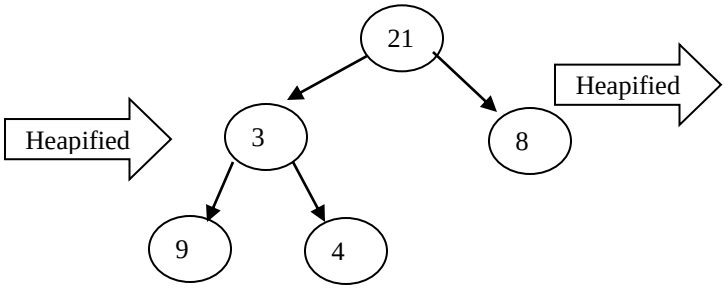


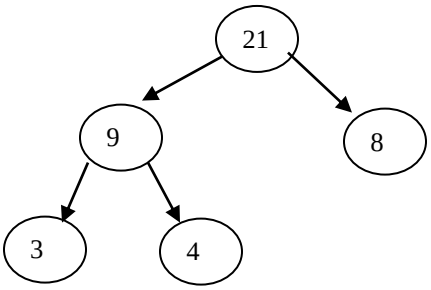
Delete root node i.e. 23. Swap root element and last element of tree



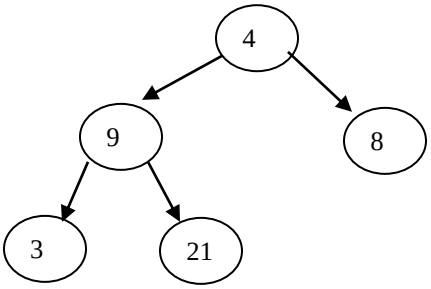
3	21	8	9	4	**23**	**36**	**39**	**67**



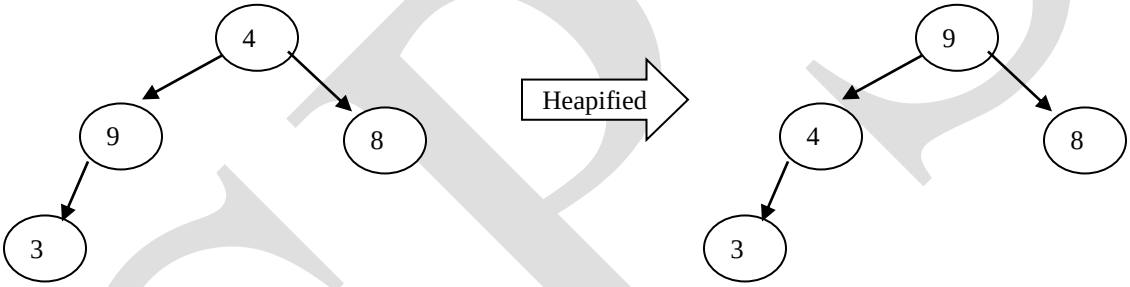




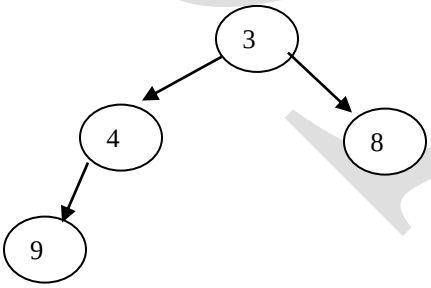
Delete root node i.e. 21. Swap root element and last element of tree



4	9	8	3	**21**	**23**	**36**	**39**	**67**

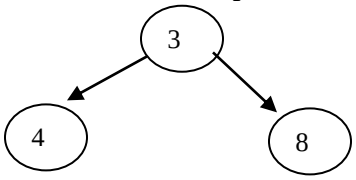


Delete root node i.e. 9. Swap root element and last element of tree

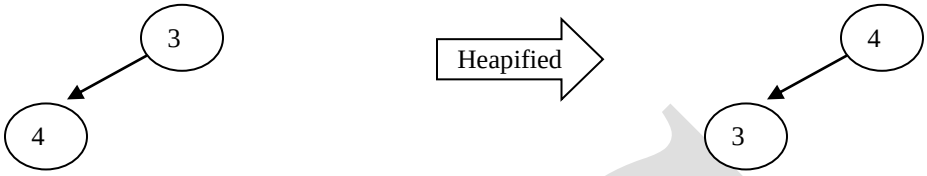


3	4	8	**9**	**21**	**23**	**36**	**39**	**67**

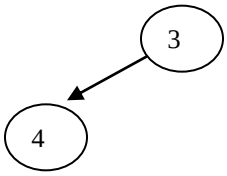
Delete root node i.e. 8. Swap root element and last element of tree



3	4	**8**	**9**	**21**	**23**	**36**	**39**	**67**



Delete root node i.e. 4. Swap root element and last element of tree



3	**4**	**8**	**9**	**21**	**23**	**36**	**39**	**67**

Delete root node i.e. 3. Swap root element and last element of tree

3	**4**	**8**	**9**	**21**	**23**	**36**	**39**	**67**

Hence we get sorted element as 3 4 8 9 21 23 36 39 67

Searching

A table of a file is a group of elements, each of which is called a record. Associated with each record a key is used to differentiate records. Key is contained within the record as a specific affect. For every file there is at least one set of key that is unique such a key is called primary key.

A searching algorithm is an algorithm that accepts key. It is possible that the search for a particular record in a table is argument as its key. If no record is found them the algorithm may return a special 'null record'. A successful search is often called retrieval. A table of records in which a key is stored for retrieval is often called a search table.

Sequential Search / Linear Search-

This is our first searching method. It is simplest way to do a search. It begin at first record and scan down, until the desired key is found or reached at the end. Generally we use this method for small list.

In sequential search we start searching for the target at the beginning of the list. We compare target with first element in the list. If match not found then compare with next element in the list. This process is continue until we find the target or we reach at the end of list.

Example-

Consider an array of 5 elements-

i.e.	10	20	30	40	50
	0	1	2	3	4

1. The element to be searched KEY=30 each element of the array is compared with KEY=30 match is found at position 2 & hence the searching is completed.
2. The element to be searched KEY=25. Each element of the array is compared with KEY=25 match is not found at in the array> Hence error message "No such element in the list."

Algorithm

1. START
2. INPUT N
3. INPUT N numbers
4. INPUT target
5. for i=0 to n-1 do
   if a[i] = target then
   print "Element found at i^{th} index" & go to step 7
6. print "Element not present"
7. STOP

Advantages

1. It is simple to implement
2. Sequential search is performed on smaller index table, hence complexity will be reduced
3. It is not required specific ordering of data

Disadvantages-

1. It is very slow process.
2. If table size is too large then use of index does not achieve enough efficiently.
3. Insertion into index table is another difficult tasks

Binary Search

Binary search is an extremely efficient algorithm as compare to linear/ Sequential search. Binary Search is searching a record in sequential table without use of auxiliary index or tables. Binary search takes a sorted array/ index as input.

In binary search we first compare the key or target with the middle element of the array. If match found then we stop, otherwise it divides the array into 2 subarrays. Then compare target with middle element, if the target is less than the middle element of the array then the element must be present in left subarray, otherwise in right subarray. Repeat this process for left or right subarray. In this way at each step we reduce the length of search by half hence we name binary search.

To find middle position use mid = (low + hi) / 2

Example

Consider the following set of numbers-

10 20 30 40 50 60 70 80

(i) 30 (ii) 50 using binary search

Low	High	Mid-position	Remarks	
0	7	3	30<40	∴ high= mid- 1
0	2	1	30 > 20	∴ low=mid +1
2	2	2	key found	
0	7	3	50>40,	∴ low=mid +1
4	7	5	50 < 60,	∴ high=mid -1
4	4	4	key found	

Algorithm

1. START
2. Input n
3. Read ‘n’ elements in an array.
4. Read the element to be searched, say ‘KEY’.
5. low=0, high=n-1
6. Find the middle element, mid = (low + high) /2
7. If (KEY=a[mid]) then display “Element found & it’s location is ‘mid’” & go to step 11.
8. If (KEY < a[mid]), then search to be continued in low=low; high=mid-1 & go to step 10.
9. If (KEY > a[mid]), then search to be continued in low = mid + 1; high=high
10. If (high ≥ low) then goto step 5 else display (“Element not in the list”).
11. STOP

C Program

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

void main()
{
```

```
        int list[],target,index,n,i;
        printf("enter size of an array: ");
        scanf("%d",&n);
        printf("\n please Enter numbers in ascending sorted order:");
    for (i = 0; i < n; i++)
        {
        scanf("%d",&list[i]);
    }

        printf("\nArray elements are as follows: ");
        for(i=0;i<n;i++)
        {
                printf("%d ",list[i]);
        }


    printf("\nEnter a number to search for: ");
    scanf("%d",&target);

    index = BSearch(list,target,0,n-1);

        if (index != -1)
        {
        printf("\nFound at index: %d",index);
    }
        else
        {
        printf("\nNot Found");
    }

        getch();
}


int BSearch(int[] list, int target,int s,int e)
{
        int start, end, mid;
    start = s;
    end = e;
    while (start <= end)
        {
        mid = (start + end) / 2;

                if (list[mid] == target)
                {
                return mid;
            }
                else if (list[mid] < target)
                {
```

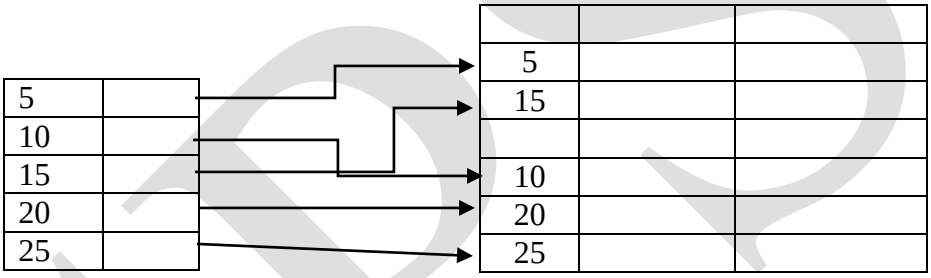
```
                start = mid + 1;
            }
                else
                {
                end = mid - 1;
            }
        }

        return -1;
}
```

Index Sequential Search

In index sequential search we are maintain two files

a. **A sorted index file:** In index file we stored a key element with offset of that particular record (Pointer to record) is stored
b. **Normal sequential file:** In sequential file our actual data (Record) is stored.

.



Index file **Sequential File**

Algorithm

1. START
2. Input N Number
3. Input N Key (K) and N Record (R)
4. Input target
5. For i=0 to n-1 perform following steps
6. if target = k[i] then print “Target found” & access the record associated with key target & go to step 9
7. i= i+1 & go to step 6
8. if i>n-1 then print “Record not found”.
9. STOP

C Program

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int arr[10],size,i,n,num;
        clrscr();
        printf("\nEnter size of array: ");
```

```
	scanf("%d",&size);

	printf("\nEnter numbers");

	for(i = 0; i < size; i++)
	{
		scanf("%d",&arr[i]);
	}
	printf("\narray elements are: ");
	for(i=0;i<size;i++)
	{
		printf("%d ",arr[i]);
	}

	printf("\nEnter the element to search: ");
	scanf("%d",&num);

	n = find(arr,num,size);

	if ((n >= 0) && (n < size))
	{
		printf("Found at index: %d",n);
	}
	else
	{
		printf("\nNot Found");
	}
	getch();
}

int find(int list[], int target,int last)
{
	int i;
	for (i = 0; i < last; i++)
	{
		if(list[i]==target)
			return i;
	}
	return -1;

}
```

Hashing

Hashing is an effective way to reduce the number of comparison. Actually hashing deals with idea of providing direct address of record where record likely to be stored. Hash Table is a data structure which store data in associative manner. In hash table, data is stored in array format where each data values has its own unique index value. Access of data becomes very fast if we know the index of desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of size of data. Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from.

Hash Search:

The goal of hash search is to find data with only one test or comparison. Hash search is a search in which the key of algorithm function is evaluated to determine location of data. Since we are searching an array we use a hashing algorithm to transfer the key into an index that contain a data.

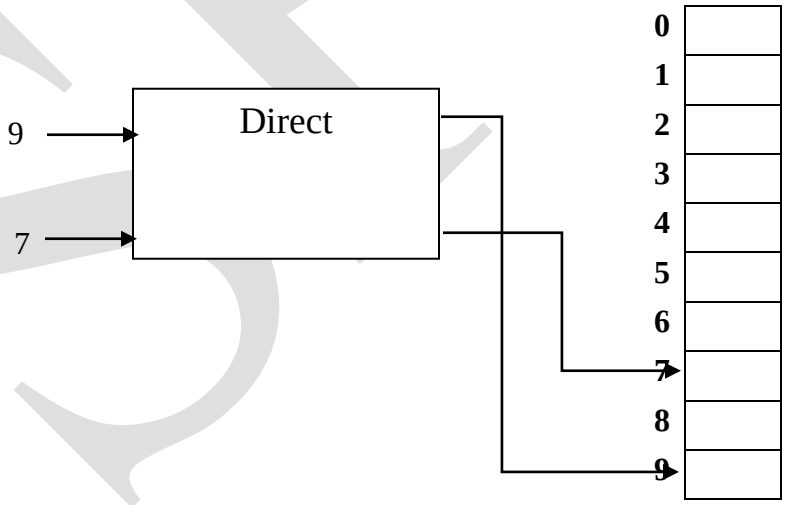
Hashing Method

1. **Direct Hashing**

   In direct Hashing the key is an address without any algorithmic penetration. Therefore the data must contain an element for every possible key.

   Address = key

   While the situation where you can use directly hashing are limited. It is used for small record but it is very powerful because it guaranties that there will be no collision.

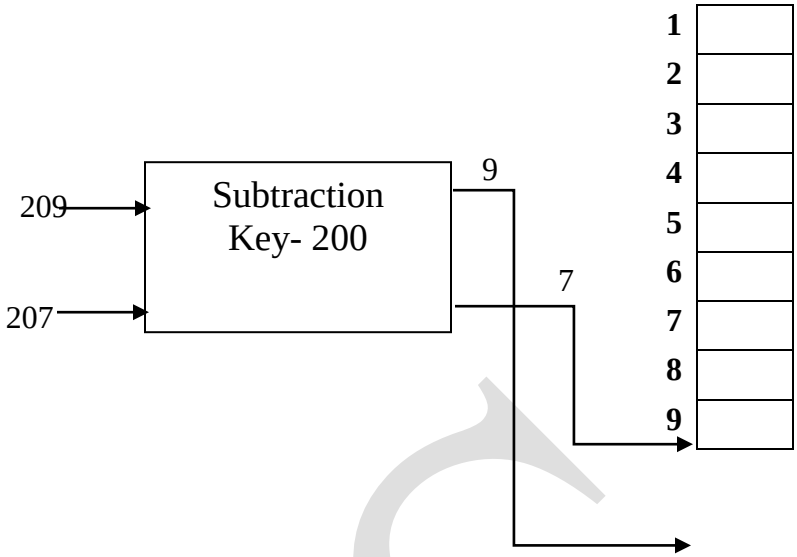


h(7)= 7
h(9) = 9

2. **Subtraction Hashing**

   Sometimes we have keys that have consecutive but do not starts from 1. E.g. Students record that contain 70 students but its key start from 201 to 270. In this case we use very simple hashing function that subtract 200 from the key to determine the address.

   Address = key – value



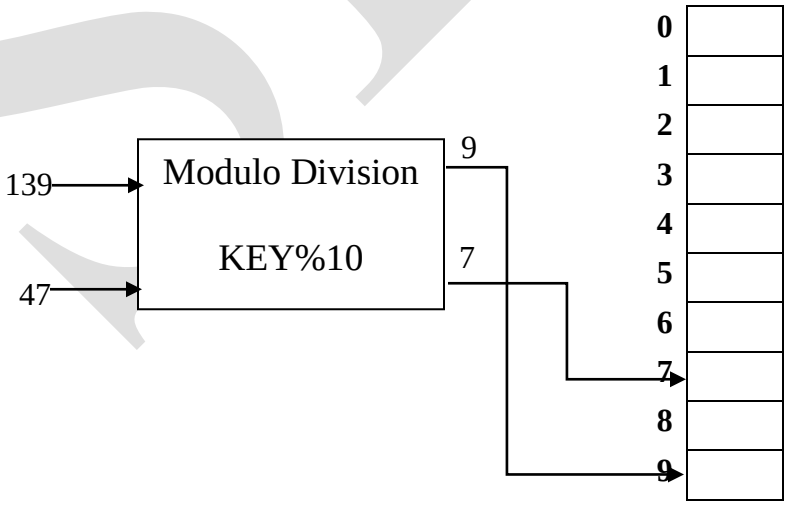
h(207)= 207 - 200 = 7
h(209) = 209 – 200 = 9

It is simple method & it also guaranties that there will be no collision.
Disadvantages:
a. It is operate only on consecutive record.
b. It is used for small records.

3. Modulo Division Method

Modulo division is also known as division reminder method. In this method divide the key by array size of the list & uses reminder for the address. This gives the simple hashing algorithm shown below where list size is the number of elements stored in array.

Address = key % List Size



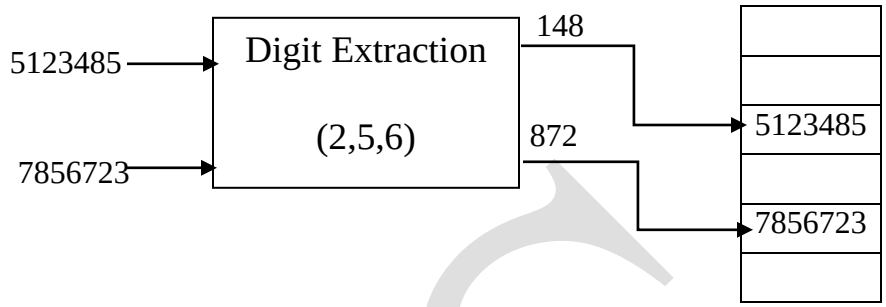
h(139)= 139 % 10 = 9
h(47) = 47 % 10 = 7

This algorithm works with only list size but when the list size is prime number then it produce less collision than other list size.

4. Digit Extraction

Using digit extraction method selected digits are extracted from the keys & use as address.

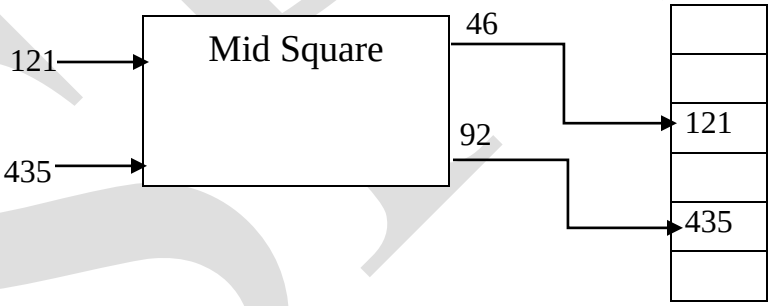
Address = Selected digits from key



5. Mid Square Hashing Method

In mid square hashing method key is square & the address is selected from middle of the square number. The limitation of this method is size of the key. i.e. suppose size of key is 6 digit then it produce square is of 12 digits, which is beyond the maximum length size of many computer.

Address = middle digits of (key)^2



h(121)= 121 * 121 = 014641 = 46
h(435) = 435 * 435 = 189225=92

Collision Resolution Method.

When we hash a new key to an address there may be possibility of collision i.e. two different key gives same address. There are several methods for handling collision each of them are independent of hashing method.

It contains following method

1. Open addressing method
    a. Linear probing
    b. Quadratic probing
    c. Double hashing
2. Linked list or separate chaining.

Linear Probing

Probe: Calculation of address & test for success is called as probe.

Linear probing is very simplest collision method. When collision is occurred then data can't be store at home address then we resolve collision by adding 1 to current location of that key. If again collision is occurred then again add 1 this process is repeat until we get blank address.

e.g. 99, 33, 23, 44, 56, 43, 19 consider hash table of size 10

h(99) = 99 % 10 = 9
h(33) = 33 % 10 = 3
h(23) = 23 % 10 = 3 → Collision
3 + 1 = 4
h(44) = 44 % 10 = 4 → Collision
4 + 1 = 5
h(56) = 56 % 10 = 6
h(43) = 43 % 10 = 3 → Collision
3 + 1 = 4
4 + 1 = 5
5 + 1 = 6
6 + 1 = 7
H(19) = 19 % 10 =9 → Collision
9 + 1 = 10%10=0

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

Advantage:

1. Simple to implement.
2. Data tends to remain nearer to home address.

2. Quadratic Probing

If collision occurred then it resolved by addressing a value other than adding 1 to the current address. Suppose a record R with key K as hash address h(key)= H then instead of searching the location with address h, h+1, h+2 …. with linearly search, search location with address h, h+1, h+4, …..

In Quadratic probe using hash function

$$H(k, i) = (h'(k) + C_1 i + C_2 i^2) \bmod m$$

Where h'= Auxiliary hash function, C_1 & $C_2 \neq 0$, Value of i = 0,1, ……., n-1 m= list size

e.g. consider inserting the key 76, 26, 37, 59, 21, 65 into hash table of size m=11 using quadratic probe with C_1= 1 & C_2 = 3 consider that the primary hash function h'(k) = k mod m

Given C_1= 1, C_2= 3 & m = 11.
h'(k) = k mod m
Consider
$H(k, i) = (h'(k) + C_1 i + C_2 i^2) \bmod m$

$H(76, 0) = (76 \bmod 11 + 1 * 0 + 3 * 0^2) \bmod 11$
$= (10 + 0 + 0) \bmod 11$
$= 10$
$H(26, 0) = (26 \bmod 11 + 1 * 0 + 3 * 0^2) \bmod 11$
$= (4 + 0 + 0) \bmod 11$
$= 4$
$H(37, 0) = (37 \bmod 11 + 1 * 0 + 3 * 0^2) \bmod 11$
$= (4 + 0 + 0) \bmod 11$
$= 4$ → Collision
$H(37, 1) = (37 \bmod 11 + 1 * 1 + 3 * 1^2) \bmod 11$
$= (4 + 1 + 3) \bmod 11$
$= 8 \bmod 11$
$= 8$
$H(59, 0) = (59 \bmod 11 + 1 * 0 + 3 * 0^2) \bmod 11$
$= (4 + 0 + 0) \bmod 11$
$= 4$ → Collision
$H(59, 1) = (59 \bmod 11 + 1 * 1 + 3 * 1^2) \bmod 11$
$= (4 + 1 + 3) \bmod 11$
$= 8 \bmod 11$
$= 8$ → Collision
$H(59, 2) = (59 \bmod 11 + 1 * 2 + 3 * 2^2) \bmod 11$
$= (4 + 2 + 12) \bmod 11$
$= 18 \bmod 11$
$= 7$
$H(21, 0) = (21 \bmod 11 + 1 * 0 + 3 * 0^2) \bmod 11$
$= (10 + 0 + 0) \bmod 11$
$= 10$ → Collision
$H(21, 1) = (21 \bmod 11 + 1 * 1 + 3 * 1^2) \bmod 11$
$= (10 + 1 + 3) \bmod 11$
$= 3$
$H(65, 0) = (65 \bmod 11 + 1 * 0 + 3 * 0^2) \bmod 11$
$= (10 + 0 + 0) \bmod 11$
$= 4$ → Collision
$H(65, 1) = (65 \bmod 11 + 1 * 1 + 3 * 1^2) \bmod 11$
$= (10 + 1 + 3) \bmod 11$
$= 3$ → Collision
$H(65, 2) = (65 \bmod 11 + 1 * 2 + 3 * 2^2) \bmod 11$
$= (10 + 2 + 12) \bmod 11$
$= 2$ → Collision

0	
1	
2	65
3	21
4	26
5	
6	
7	59
8	37
9	
10	76

3. Double Hashing

Double hashing is technique in which a second hash function is applied to the key when a collision occurs. When collision is occurred then you have to apply second hash function. There are two important rules to be followed for second function.

1. It never evaluate to zero.
2. Must make sure that all cells can be probed.
   Double hashing used following formula

   $H_1(K) = K \bmod m$.
   $H_2(K) = R - k \bmod R$ Where R is prime number smaller than size of table

e.g. Store the values given below in array with size 11.
76, 26, 37, 59, 21, 66.

$H_1(76) = 76 \% 11 = 10$

$H_1(26) = 26 \% 11 = 4$

$H_1(37) = 37 \% 11 = 4 \rightarrow$ collision
$H_2(37) = 7 - 37 \% 7 = 7 - 2 = 5$
Jump 5 position from index 4 i.e. place 37 at index 9

$H_1(59) = 59 \% 11 = 4 \rightarrow$ collision
$H_2(59) = 7 - 59 \% 7 = 7 - 3 = 4$
Jump 4 position from index 4 i.e. place 59 at index 8

$H_1(21) = 21 \% 11 = 10 \rightarrow$ collision
$H_2(21) = 7 - 21 \% 7 = 7 - 0 = 7$
Jump 7 position from index 10 i.e. place 21 at index 6

$H_1(66) = 66 \% 11 = 0$

0	66
1	
2	
3	
4	26
5	
6	21
7	
8	59
9	37
10	76

B Chaining or Linked List

In chaining each location in hash table stores a pointer to a linked list that contains all the key values of hash having same location

e.g. Insert keys 7, 24, 18, 52, 36, 54, 11, 23 & 16 in chain hash table o 9 memory location use modular hash function.

H(7) = k mod m
= 7 mod 9 = 7

H(24) = k mod m
= 24 mod 9 = 6

H(18) = k mod m
= 18 mod 9 = 0

H(52) = k mod m
= 52 mod 9 = 7

H(36) = k mod m
= 36 mod 9 = 0

H(54) = k mod m
= 54 mod 9 = 0

H(11) = k mod m
= 11 mod 9 = 2

H(23) = k mod m
= 23 mod 9 = 5

H(16) = k mod m
= 16 mod 9 = 7