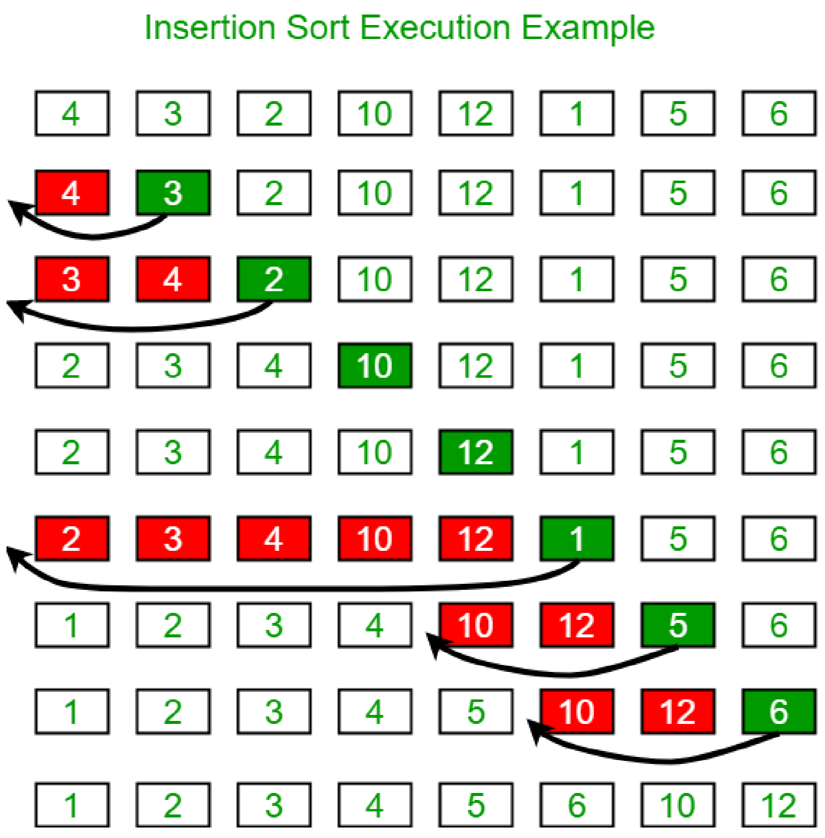
Insertion Sort

Insertion Sort *is a simplest data Sorting algorithm which sorts the array elements by shifting elements one by one and inserting each element into its proper position. This technique is also used for sort array elements. With the help of below animated image you can easily understand and you can also see real life example in second image.*



Advantage of this sorting technique

It is a simple data Sorting algorithm. It is also better than Selection Sort and Bubble Sort algorithms.

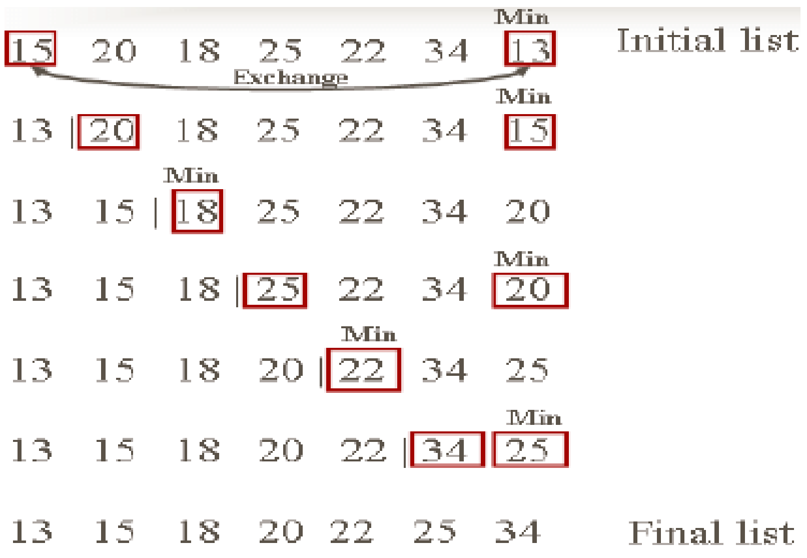
Dis-Advantage of this sorting technique

This sorting technique is slower than quick sort sorting algorithm.

Selection Sort

Selection sort is based of maximum and minimum value. First check minimum value in array list and place it at first position (position 0) of array, next find second smallest element in array list and place this value at second position (position 1) and so on. Same process is repeated until sort all element of an array.

- Find the minimum element in the list.
- Swap it with the element in the first position of the list.
- Repeat the steps above for all remaining elements of the list starting from the second position.



Advantage of this sorting technique

It is very simple and easy method to sort elements.

Dis-Advantage of this sorting technique

It is time consuming and slow process to sort elements.

Counting Sort

Counting Sort Algorithm is an efficient sorting algorithm that can be used for sorting elements within a specific range. This sorting technique is based on the frequency/count of each element to be sorted and works using the following algorithm-

- **Input**: Unsorted array A[] of n elements
- **Output**: Sorted arrayB[]

Step 1: Consider an input array A having n elements in the range of 0 to k, where n and k are positive integer numbers. These n elements have to be sorted in ascending order using the counting sort technique. Also note that A[] can have distinct or duplicate elements

Step 2: The count/frequency of each distinct element in A is computed and stored in another array, say count, of size k+1. Let u be an element in A such that its frequency is stored at count[u].

Step 3: Update the count array so that element at each index, say i, is equal to -

Step 4: The updated count array gives the index of each element of array A in the sorted sequence. Assume that the sorted sequence is stored in an output array, say B, of size n.

Step 5: Add each element from input array A to B as follows:

a. Set i=0 and t = A[i]

b. Add t to B[v] such that v = (count[t]-1).

c. Decrement count[t] by 1

d. Increment i by 1

Repeat steps **(a)** to **(d)** till **i = n-1**

Step 6: Display B since this is the sorted array

Pictorial Representation of Counting Sort with an Example

Let us trace the above algorithm using an example:

Consider the following input array A to be sorted. All the elements are in range 0 to 9

```
A[]= {1, 3, 2, 8, 5, 1, 5, 1, 2, 7}
```

Step 1: Initialize an auxiliary array, say count and store the frequency of every distinct element. Size of count is 10 (k+1, such that range of elements in A is 0 to k)

Figure 1: count array

Step 2: Using the formula, updated count array is -

Figure 2: Formula for updating count array

0	1	2	3	4	5	6	7	8	9	
0	3	5	6	6	8	8	9	10	10	count

Figure 3 : Updated count array

Step 3: Add elements of array A to resultant array B using the following steps:

- For, i=0, t=1, count[1]=3, v=2. After adding 1 to B[2], count[1]=2 and i=1

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	0	0	0	0	0	0	B
0	2	5	6	6	8	8	9	10	10	count

Figure 4: For i=0

- For i=1, t=3, count[3]=6, v=5. After adding 3 to B[5], count[3]=5 and i=2

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	0	3	0	0	0	0	B
0	2	5	5	6	8	8	9	10	10	count

Figure 5: For i=1

- For i=2, t=2, count[2]= 5, v=4. After adding 2 to B[4], count[2]=4 and i=3

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	2	3	0	0	0	0	B
0	2	4	5	6	8	8	9	10	10	count

Figure 6: For i=2

- For i=3, t=8, count[8]= 10, v=9. After adding 8 to B[9], count[8]=9 and i=4

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	2	3	0	0	0	8	B
0	2	4	5	6	8	8	9	9	10	count

Figure 7: For i=3

- *On similar lines, we have the following:*

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	2	3	0	5	0	8	B
0	2	4	5	6	7	8	9	9	10	count

Figure 8: For i=4

0	1	2	3	4	5	6	7	8	9	
0	1	1	0	2	3	0	5	0	8	B
0	1	4	5	6	7	8	9	9	10	count

Figure 9: For i=5

0	1	2	3	4	5	6	7	8	9	
0	1	1	0	2	3	5	5	0	8	B
0	1	4	5	6	6	8	9	9	10	count

Figure 10: For i=6

0	1	2	3	4	5	6	7	8	9	
1	1	1	0	2	3	5	5	0	8	B
0	0	4	5	6	6	8	9	9	10	count

Figure 11: For i=7

0	1	2	3	4	5	6	7	8	9	
1	1	1	2	2	3	5	5	0	8	B
0	0	3	5	6	6	8	9	9	10	count

Figure 12: For i=8

0	1	2	3	4	5	6	7	8	9	
1	1	1	2	2	3	5	5	7	8	B
0	0	3	5	6	6	8	8	9	10	count

Figure 13: For i=9

Thus, array B has the sorted list of elements.

Radix Sort

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

Step by Step Process

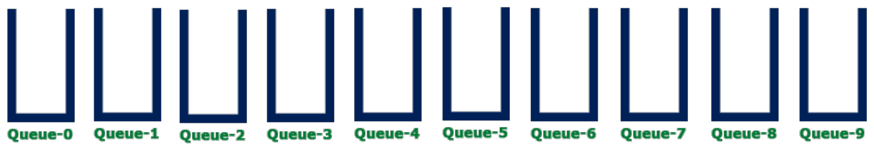
The Radix sort algorithm is performed using the following steps...

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

Consider the following list of unsorted integer numbers

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

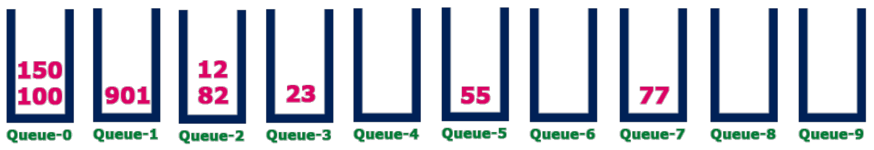
Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.



Pass - 1

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

8**2**, 90**1**, 10**0**, 1**2**, 15**0**, 7**7**, 5**5** & 2**3**



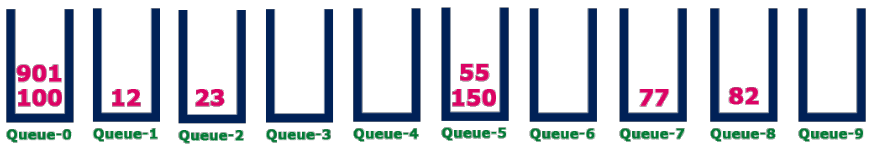
Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

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

Pass - 2

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

1**0**0, 1**5**0, 9**0**1, **8**2, **1**2, **2**3, **5**5 & **7**7



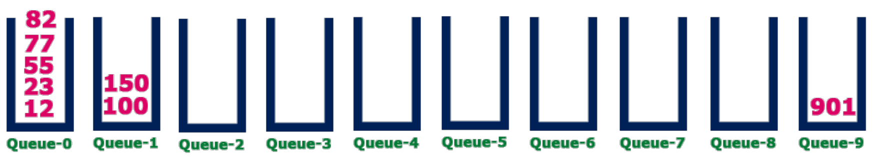
Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

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

Pass - 3

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

100, **9**01, 12, 23, **1**50, 55, 77 & 82



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

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

List got sorted in the incresing order.

Shell sort

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as **interval**. This interval is calculated based on Knuth's formula as −

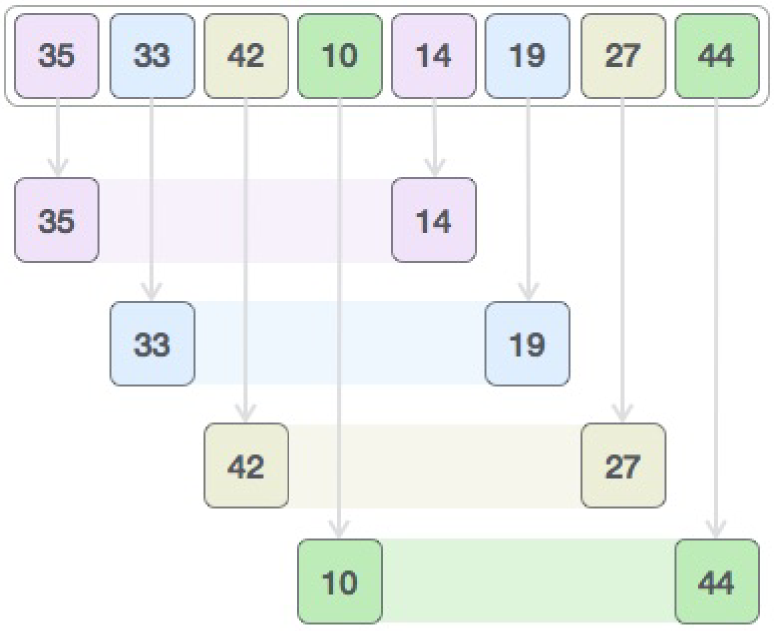
Knuth's Formula

h = h * 3 + 1
where −
  h is interval with initial value 1

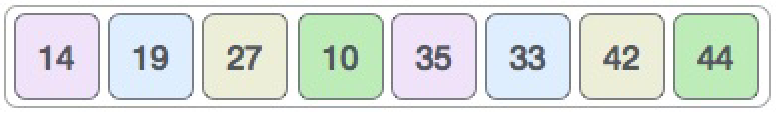
This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is (n), where n is the number of items. And the worst case space complexity is O(n).

How Shell Sort Works?

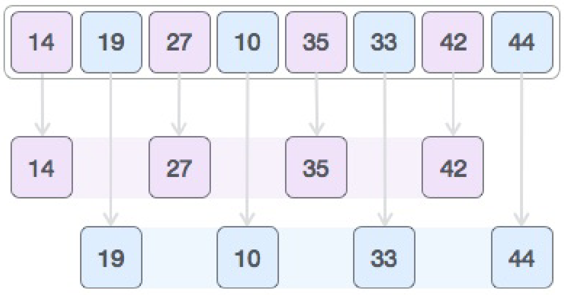
Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



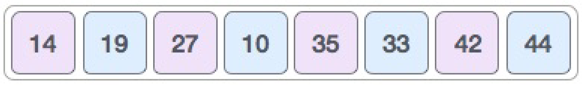
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −



Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}

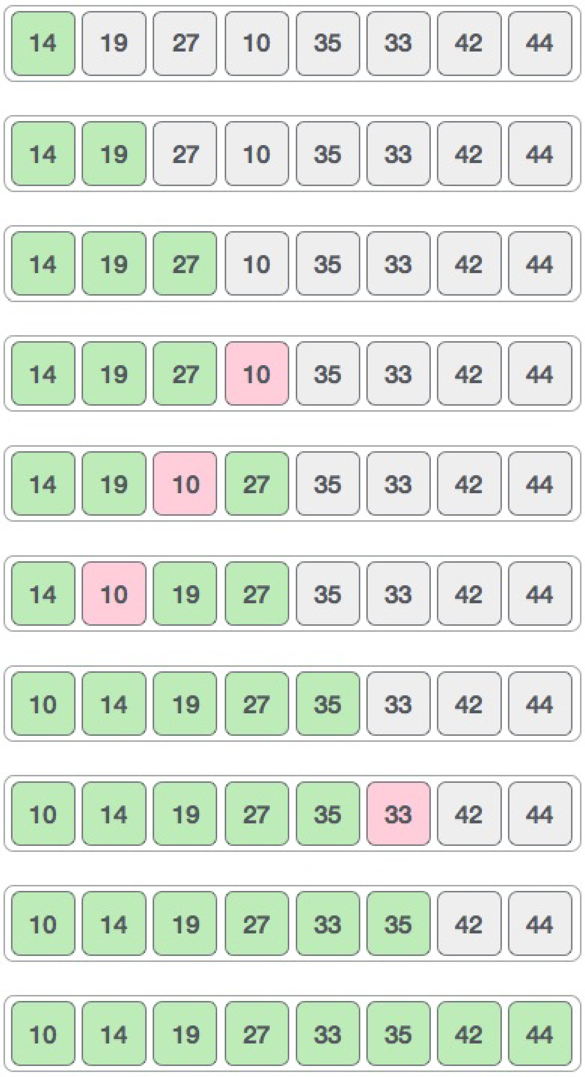


We compare and swap the values, if required, in the original array. After this step, the array should look like this −



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction −



We see that it required only four swaps to sort the rest of the array.

Algorithm

Following is the algorithm for shell sort.

Step 1 − Initialize the value of *h*
Step 2 − Divide the list into smaller sub-list of equal interval *h*
Step 3 − Sort these sub-lists using **insertion sort**
Step 3 − Repeat until complete list is sorted

Merge Sort

Divide and Conquer

If we can break a single big problem into smaller sub-problems, solve the smaller sub-problems and combine their solutions to find the solution for the original big problem, it becomes easier to solve the whole problem.

Let's take an example, **Divide and Rule**.

When Britishers came to India, they saw a country with different religions living in harmony, hard working but naive citizens, unity in diversity, and found it difficult to establish their empire. So, they adopted the policy of **Divide and Rule**. Where the population of India was collectively a one big problem for them, they divided the problem into smaller problems, by instigating rivalries between local kings, making them stand against each other, and this worked very well for them.

Well that was history, and a socio-political policy (**Divide and Rule**), but the idea here is, if we can somehow divide a problem into smaller sub-problems, it becomes easier to eventually solve the whole problem.

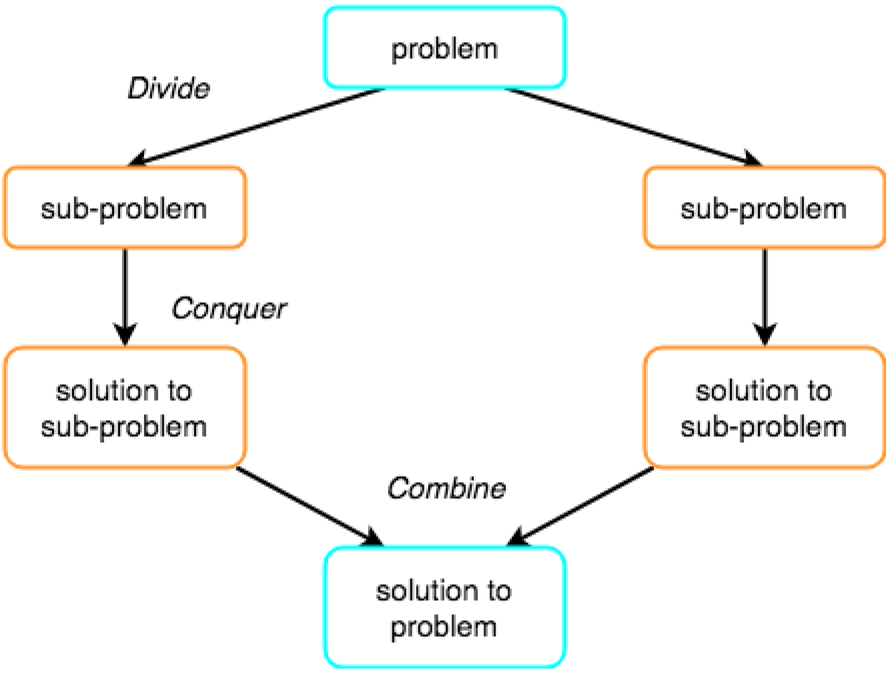
In **Merge Sort**, the given unsorted array with n elements, is divided into n subarrays, each having **one** element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

1. **Divide** the problem into multiple small problems.

2. **Conquer** the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.

3. **Combine** the solutions of the subproblems to find the solution of the actual problem.



How Merge Sort Works?

As we have already discussed that merge sort utilizes divide-and-conquer rule to break the problem into sub-problems, the problem in this case being, **sorting a given array**.

In merge sort, we break the given array midway, for example if the original array had 6 elements, then merge sort will break it down into two subarrays with 3 elements each.

But breaking the orignal array into 2 smaller subarrays is not helping us in sorting the array.

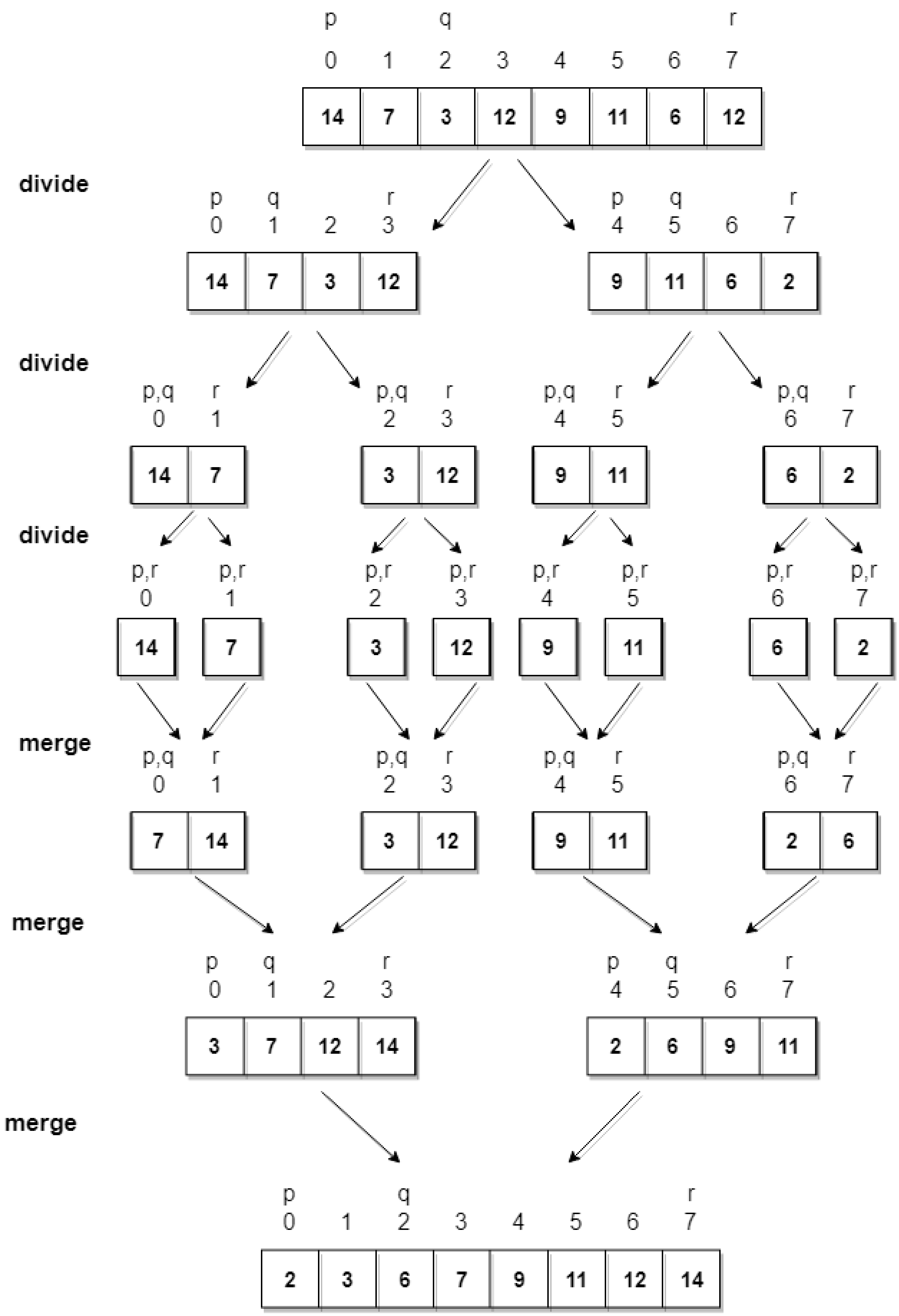
So we will break these subarrays into even smaller subarrays, until we have multiple subarrays with **single element** in them. Now, the idea here is that an array with a single element is already sorted, so once we break the original array into subarrays which has only a single element, we have successfully broken down our problem into base problems.

And then we have to merge all these sorted subarrays, step by step to form one single sorted array.

```
MergeSort(A, p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)
```

Let's consider an array with values {14, 7, 3, 12, 9, 11, 6, 12}

Below, we have a pictorial representation of how merge sort will sort the given array.


p q r
0 1 2 3 4 5 6 7
14 7 3 12 9 11 6 12
divide
p q r
0 1 2 3
14 7 3 12
p q r
4 5 6 7
9 11 6 2
divide
p,q r
0 1
14 7
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
6 2
divide
p,r 0 14
p,r 1 7
p,r 2 3
p,r 3 12
p,r 4 9
p,r 5 11
p,r 6 6
p,r 7 2
merge
p,q r
0 1
7 14
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
2 6
merge
p q r
0 1 2 3
3 7 12 14
p q r
4 5 6 7
2 6 9 11
merge
p q r
0 1 2 3 4 5 6 7
2 3 6 7 9 11 12 14

In merge sort we follow the following steps:

1. We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.

2. Then we find the middle of the array using the formula $(p + r)/2$ and mark the middle index as q, and break the array into two subarrays, from p to q and from q + 1 to r index.

3. Then we divide these 2 subarrays again, just like we divided our main array and this continues.

4. Once we have divided the main array into subarrays with single elements, then we start merging the subarrays.

Quicksort

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

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

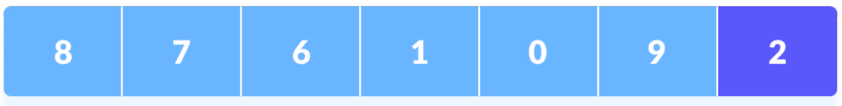
   While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
2. The left and right subarrays are also divided using the same approach. This process continues until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

3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Working of Quicksort Algorithm

1. Select the Pivot Element

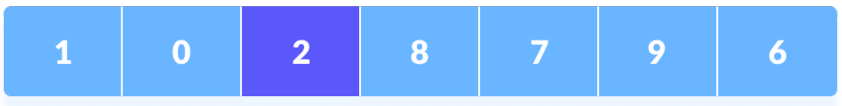
There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.



Select a pivot element

2. Rearrange the Array

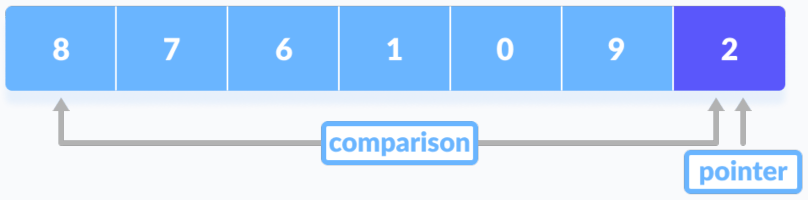
Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.



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

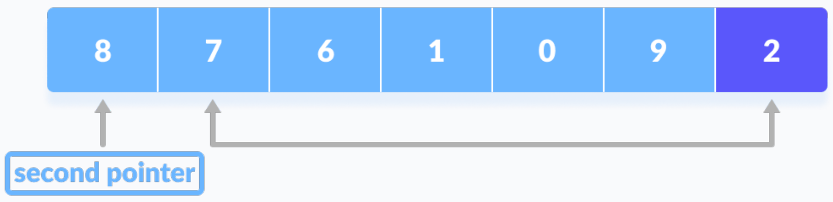
Here's how we rearrange the array:

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



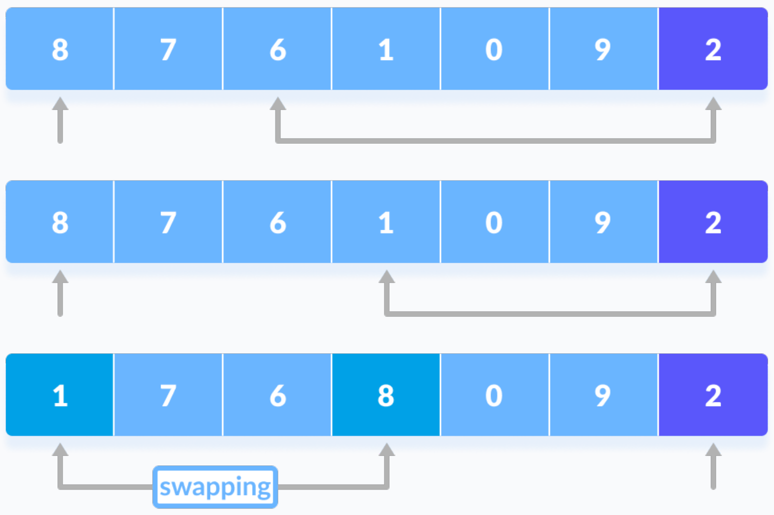
2. Comparison of pivot element with element beginning from the first index

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



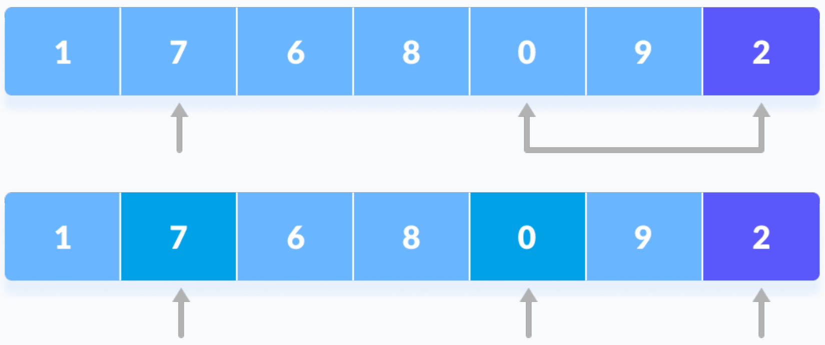
4. If the element is greater than the pivot element, a second pointer is set for that element.

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



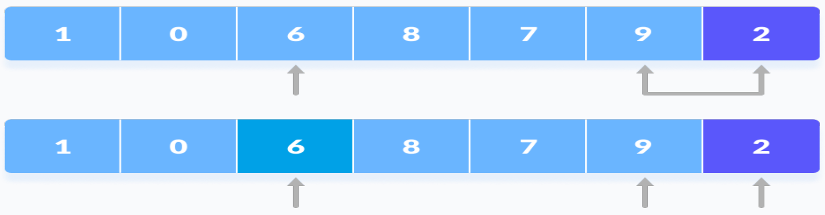
6. Pivot is compared with other elements.

7. Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.



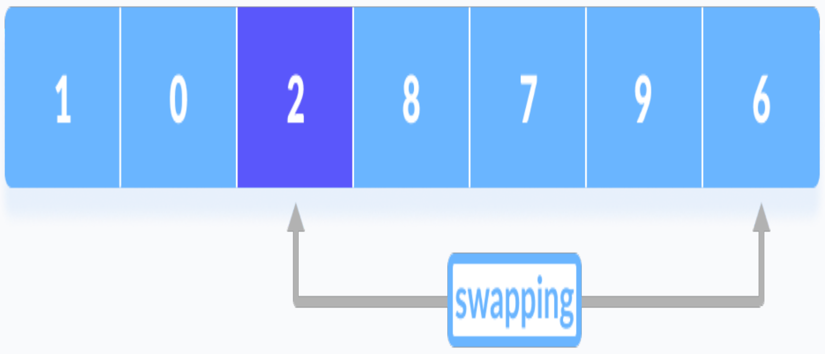
The process is repeated to set the next greater element as the second pointer.

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



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

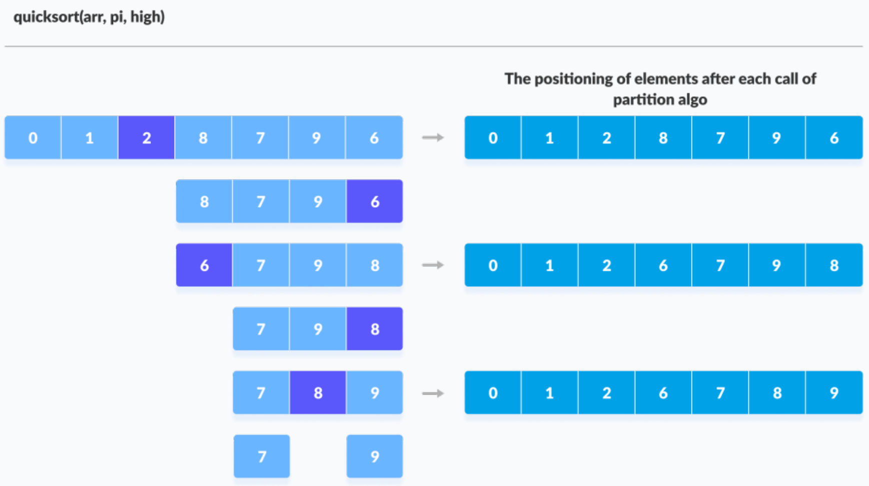
10. Finally, the pivot element is swapped with the second pointer.



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

3. Divide Subarrays

Pivot elements are again chosen for the left and the right sub-parts separately. And, **step 2** is repeated.


quicksort(arr, pi, high)
The positioning of elements after each call of partition algo

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

Analysis of different sorting techniques

Comparison based sorting –
In comparison based sorting, elements of an array are compared with each other to find the sorted array.

- Bubble sort and Insertion sort –
  Average and worst case time complexity: n^2
  Best case time complexity: n when array is already sorted.
  Worst case: when the array is reverse sorted.

- Selection sort –
  Best, average and worst case time complexity: n^2 which is independent of distribution of data.

- Merge sort –
  Best, average and worst case time complexity: nlogn which is independent of distribution of data.

,

- Quicksort –
  It is a divide and conquer approach with recurrence relation:

$T(n) = T(k) + T(n-k-1) + cn$

- Worst case: when the array is sorted or reverse sorted, the partition algorithm divides the array in two subarrays with 0 and n-1 elements. Therefore,

$T(n) = T(0) + T(n-1) + cn$

Solving this we get, $T(n) = O(n^2)$

- Best case and Average case: On an average, the partition algorithm divides the array in two subarrays with equal size. Therefore,

$T(n) = 2T(n/2) + cn$

Solving this we get, $T(n) = O(n\log n)$

Non-comparison based sorting –
In non-comparison based sorting, elements of array are not compared with each other to find the sorted array.

- Radix sort –
  Best, average and worst case time complexity: nk where k is the maximum number of digits in elements of array.

- Count sort –
  Best, average and worst case time complexity: n+k where k is the size of count array.

In-place/Outplace technique –

A sorting technique is inplace if it does not use any extra memory to sort the array.
Among the comparison based techniques discussed, only merge sort is outplaced technique as it requires an extra array to merge the sorted subarrays.
Among the non-comparison based techniques discussed, all are outplaced techniques. Counting sort uses a counting array and bucket sort uses a hash table for sorting the array.
Online/Offline technique –

A sorting technique is considered Online if it can accept new data while the procedure is ongoing i.e. complete data is not required to start the sorting operation.
Among the comparison based techniques discussed, only Insertion Sort qualifies for this because of the underlying algorithm it uses i.e. it processes the array (not just elements) from left to right and if new elements are added to the right, it doesn't impact the ongoing operation.

Stable/Unstable technique –

A sorting technique is stable if it does not change the order of elements with the same value.
Out of comparison based techniques, bubble sort, insertion sort and merge sort are stable techniques. Selection sort is unstable as it may change the order of elements with the same value. For example, consider the array 4, 4, 1, 3.
In the first iteration, the minimum element found is 1 and it is swapped with 4 at 0th position. Therefore, the order of 4 with respect to 4 at the 1st position will change.
Similarly, quick sort and heap sort are also unstable.

Out of non-comparison based techniques, Counting sort and Bucket sort are stable sorting techniques whereas radix sort stability depends on the underlying algorithm used for sorting.

Analysis of sorting techniques :

- *When the array is almost sorted, insertion sort can be preferred.*
- *When order of input is not known, merge sort is preferred as it has worst case time complexity of nlogn and it is stable as well.*
- *When the array is sorted, insertion and bubble sort gives complexity of n but quick sort gives complexity of n^2.*

Time and Space Complexity Comparison Table :

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	*(N)*	*(N^2)*	*$O(N^2)$*	*O(1)*
Selection Sort	*(N^2)*	*(N^2)*	*$O(N^2)$*	*O(1)*
Insertion Sort	*(N)*	*(N^2)*	*$O(N^2)$*	*O(1)*

Sorting Algorithm	Time Complexity			Space Complexity
Merge Sort	(N log N)	(N log N)	O(N log N)	O(N)
Heap Sort	(N log N)	(N log N)	O(N log N)	O(1)
Quick Sort	(N log N)	(N log N)	$O(N^2)$	O(log N)
Radix Sort	(N k)	(N k)	O(N k)	O(N + k)
Count Sort	(N + k)	(N + k)	O(N + k)	O(k)
Bucket Sort	(N + k)	(N + k)	$O(N^2)$	O(N)

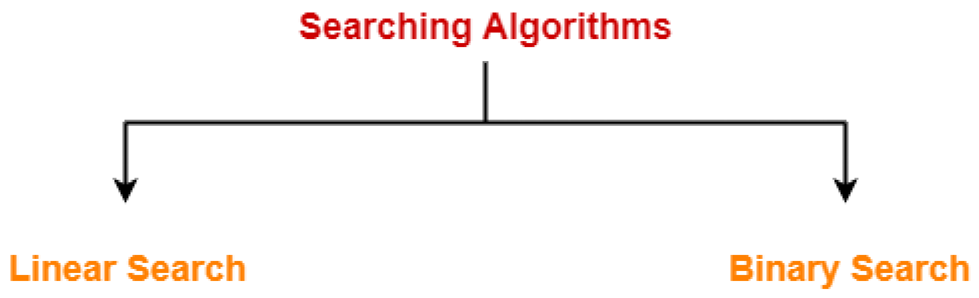
Searching

Searching-

- Searching is a process of finding a particular element among several given elements.
- The search is successful if the required element is found.
- Otherwise, the search is unsuccessful.

Searching Algorithms-

The searching of an element in the given array may be carried out in the following two ways-



1. Linear Search
2. Binary Search

Linear Search:

- Linear Search is the simplest searching algorithm.
- It traverses the array sequentially to locate the required element.
- It searches for an element by comparing it with each element of the array one by one.
- So, it is also called as **Sequential Search**.

Linear Search Algorithm is applied when-

- No information is given about the array.
- The given array is unsorted or the elements are unordered.
- The list of data items is smaller.

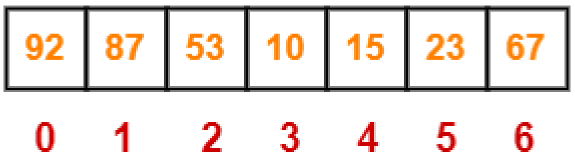
A simple approach to implement a linear search is

- Begin with the leftmost element of arr[] and one by one compare x with each element.
- If x matches with an element then return the index.
- If x does not match with any of the elements then return -1.

Linear Search Example-

Consider-

- We are given the following linear array.
- Element 15 has to be searched in it using Linear Search Algorithm.



LInear Search Example

Now,

- Linear Search algorithm compares element 15 with all the elements of the array one by one.
- It continues searching until either the element 15 is found or all the elements are searched.

Linear Search Algorithm works in the following steps-

Step-01:

- It compares element 15 with the 1st element 92.
- Since 15 ≠ 92, so required element is not found.
- So, it moves to the next element.

Step-02:

- It compares element 15 with the 2nd element 87.
- Since 15 ≠ 87, so required element is not found.
- So, it moves to the next element.

Step-03:

- It compares element 15 with the 3rd element 53.
- Since 15 ≠ 53, so required element is not found.

- So, it moves to the next element.

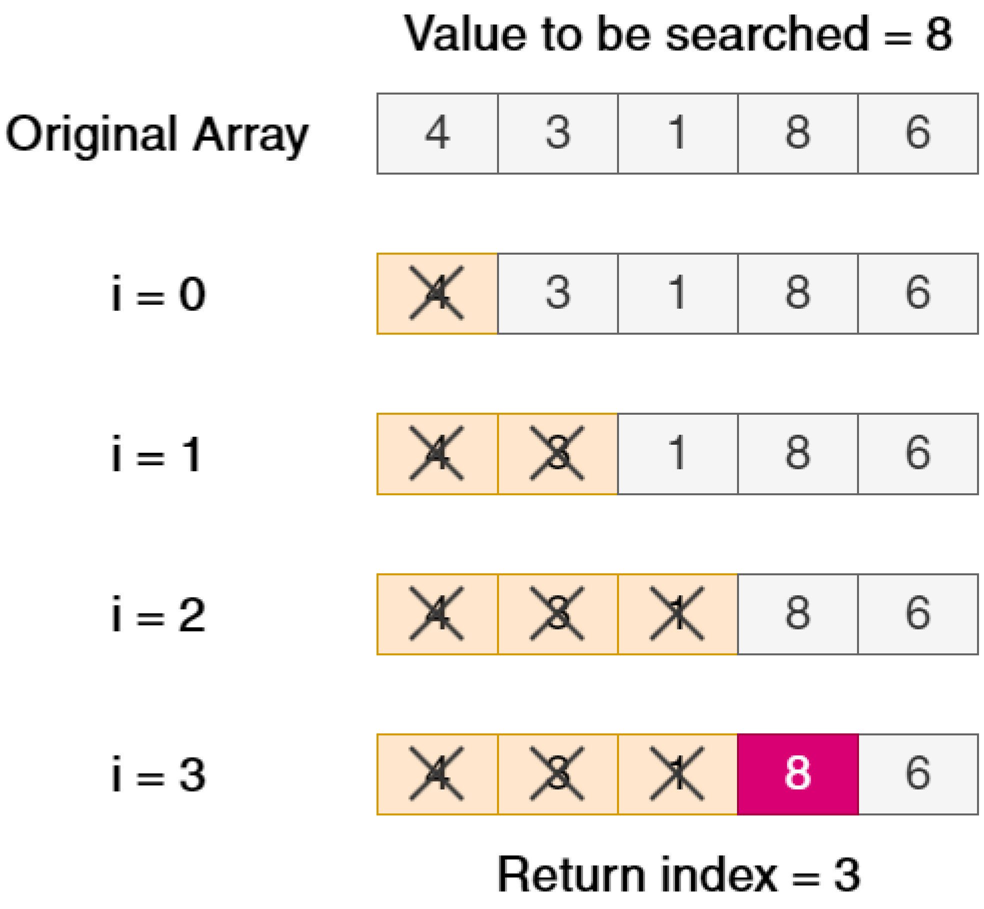
Step-04:

- It compares element 15 with the 4th element 10.
- Since 15 ≠ 10, so required element is not found.
- So, it moves to the next element.

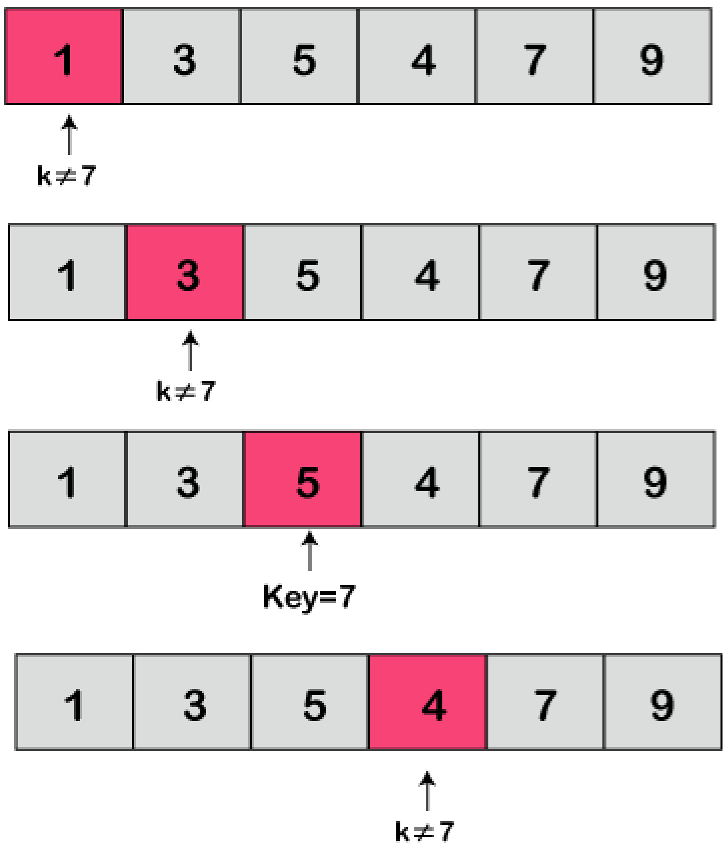
Step-05:

- It compares element 15 with the 5th element 15.
- Since 15 = 15, so required element is found.
- Now, it stops the comparison and returns index 4 at which element 15 is present.

Example2:



Example3:



<u>Time Complexity Analysis-</u>

Linear Search time complexity analysis is done below-

<u>Best case-</u>

In the best possible case,

- The element being searched may be found at the first position.
- In this case, the search terminates in success with just one comparison.
- Thus in best case, linear search algorithm takes O(1) operations.

<u>Worst Case-</u>

In the worst possible case,

- The element being searched may be present at the last position or not present in the array at all.
- In the former case, the search terminates in success with n comparisons.
- In the later case, the search terminates in failure with n comparisons.
- Thus in worst case, linear search algorithm takes O(n) operations.

Thus, we have-

Time Complexity of Linear Search Algorithm is O(n).

Here, n is the number of elements in the linear array.

<u>Linear Search Efficiency-</u>

- Linear Search is less efficient when compared with other algorithms like Binary Search & Hash tables.
- The other algorithms allow significantly faster searching.

Implementing Linear Search in C

```
#include<stdio.h>


int main()

{

    int a[20],i,x,n;

    printf("How many elements?");

    scanf("%d",&n);


    printf("Enter array elements:n");

    for(i=0;i<n;++i)

        scanf("%d",&a[i]);


    printf("nEnter element to search:");

    scanf("%d",&x);


    for(i=0;i<n;++i)

        if(a[i]==x)

            break;


    if(i<n)

        printf("Element found at index %d",i);

    else

        printf("Element not found");


    return 0;

}
```

Binary Search-

- Binary Search is one of the fastest searching algorithms.
- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquer technique.

Binary Search Algorithm can be applied only on **Sorted arrays**.

So, the elements must be arranged in-

- Either ascending order if the elements are numbers.
- Or dictionary order if the elements are strings.

To apply binary search on an unsorted array,

- First, sort the array using some sorting technique.
- Then, use binary search algorithm.

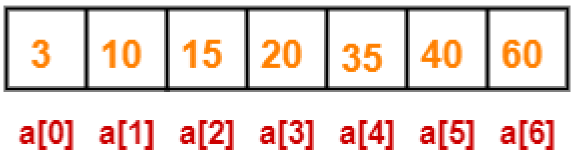
Binary search algorithm:

```
do until the pointers low and high meet each other.

    mid = (low + high)/2

    if (x == arr[mid])

        return mid

    else if (x > arr[mid]) // x is on the right side

        low = mid + 1

    else                       // x is on the left side

        high = mid - 1
```

Binary Search Example-

Consider-

- We are given the following sorted linear array.
- Element 15 has to be searched in it using Binary Search Algorithm.



Binary Search Example

Binary Search Algorithm works in the following steps-

Step-01:

- To begin with, we take beg=0 and end=6.
- We compute location of the middle element as-

mid

= (beg + end) / 2

= (0 + 6) / 2

= 3

- Here, a[mid] = a[3] = 20 ≠ 15 and beg < end.
- So, we start next iteration.

Step-02:

- Since a[mid] = 20 > 15, so we take end = mid – 1 = 3 – 1 = 2 whereas beg remains unchanged.
- We compute location of the middle element as-

mid

= (beg + end) / 2

= (0 + 2) / 2

= 1

- Here, a[mid] = a[1] = 10 ≠ 15 and beg < end.

- So, we start next iteration.

Step-03:

- Since a[mid] = 10 < 15, so we take beg = mid + 1 = 1 + 1 = 2 whereas end remains unchanged.
- We compute location of the middle element as-

mid

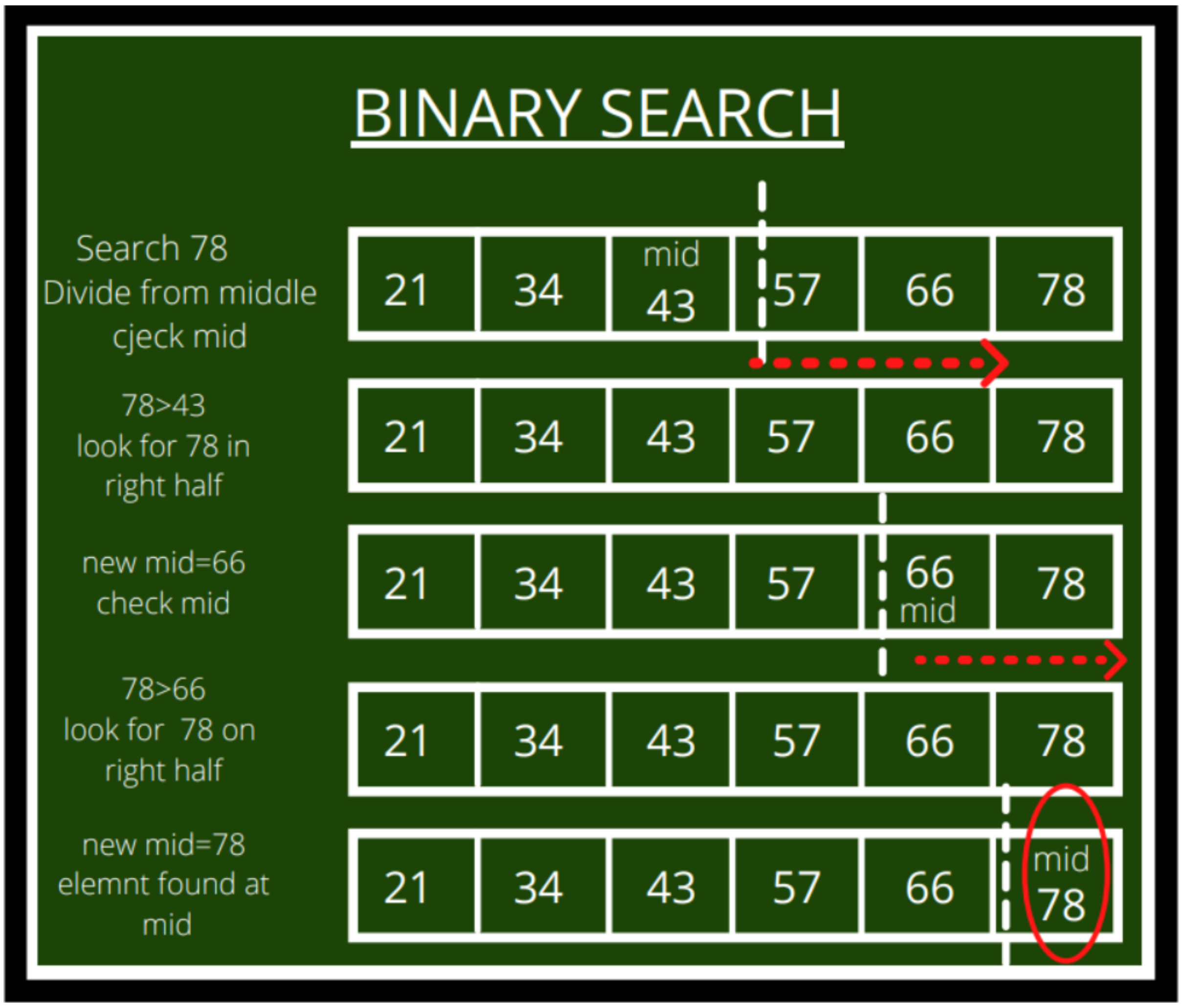
= (beg + end) / 2

= (2 + 2) / 2

= 2

- Here, a[mid] = a[2] = 15 which matches to the element being searched.
- So, our search terminates in success and index 2 is returned.

Example2:



BINARY SEARCH
Search 78 Divide from middle cjeck mid
21 34 mid 43 57 66 78
78>43 look for 78 in right half
21 34 43 57 66 78
new mid=66 check mid
21 34 43 57 66 mid 78
78>66 look for 78 on right half
21 34 43 57 66 78
new mid=78 elemnt found at mid
21 34 43 57 66 mid 78

Time Complexity Analysis-

Binary Search time complexity analysis is done below-

- In each iteration or in each recursive call, the search gets reduced to half of the array.
- So for n elements in the array, there are $\log_2 n$ iterations or recursive calls.

Thus, we have-

Time Complexity of Binary Search Algorithm is $O(\log_2 n)$.

Here, n is the number of elements in the sorted linear array.

Binary Search Algorithm Advantages-

The advantages of binary search algorithm are-

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

Binary Search Algorithm Disadvantages-

The disadvantages of binary search algorithm are-

- It employs recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor. (because of its random access nature)

Implementation:

```c
// Binary Search in C

#include <stdio.h>

int binarySearch(int array[], int x, int low, int high) {
  // Repeat until the pointers low and high meet each other
  while (low <= high) {
    int mid = low + (high - low) / 2;

    if (array[mid] == x)
      return mid;

    if (array[mid] < x)
      low = mid + 1;

    else
      high = mid - 1;
  }

  return -1;
}

int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 4;
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
  return 0;
}
```

What is Hashing?

- Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- Hashing is also known as **Hashing Algorithm** or **Message Digest Function**.
- It is a technique to convert a range of key values into a range of indexes of an array.
- It is used to facilitate the next level searching method when compared with the linear or binary search.
- Hashing allows to update and retrieve any data entry in a constant time O(1).
- Constant time O(1) means the operation does not depend on the size of the data.
- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.

What is Hash Function?

- A fixed process converts a key to a hash key is known as a **Hash Function.**
- This function takes a key and maps it to a value of a certain length which is called a **Hash value** or **Hash.**
- Hash value represents the original string of characters, but it is normally smaller than the original.
- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is transmitted without errors.

What is Hash Table?

- Hash table or hash map is a data structure used to store key-value pairs.
- It is a collection of items stored to make it easy to find them later.
- It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.

- It is an array of list where each list is known as bucket.
- It contains value based on the key.
- Hash table is used to implement the map interface and extends Dictionary class.
- Hash table is synchronized and contains only unique elements.

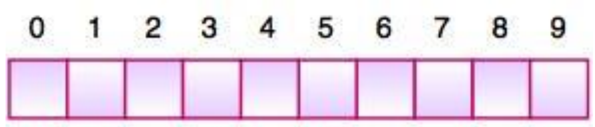


Fig. Hash Table

- The above figure shows the hash table with the size of n = 10. Each position of the hash table is called as **Slot**. In the above hash table, there are n slots in the table, names = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot 1, slot 2 and so on. Hash table contains no items, so every slot is empty.
- As we know the mapping between an item and the slot where item belongs in the hash table is called the hash function. The hash function takes any item in the collection and returns an integer in the range of slot names between 0 to n-1.
- 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

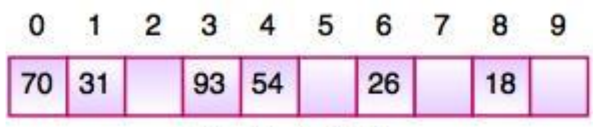


Fig. Hash Table

- After computing the hash values, we can insert each item into the hash table at the designated position as shown in the above figure. In the hash table, 6 of the 10 slots are occupied, it is referred to as the load factor and denoted by, λ = No. of items / table size. For example , λ = 6/10.
- It is easy to search for an item using hash function where it computes the slot name for the item and then checks the hash table to see if it is present.
- Constant amount of time O(1) is required to compute the hash value and index of the hash table at that location.

Collision in Hashing-

In hashing,

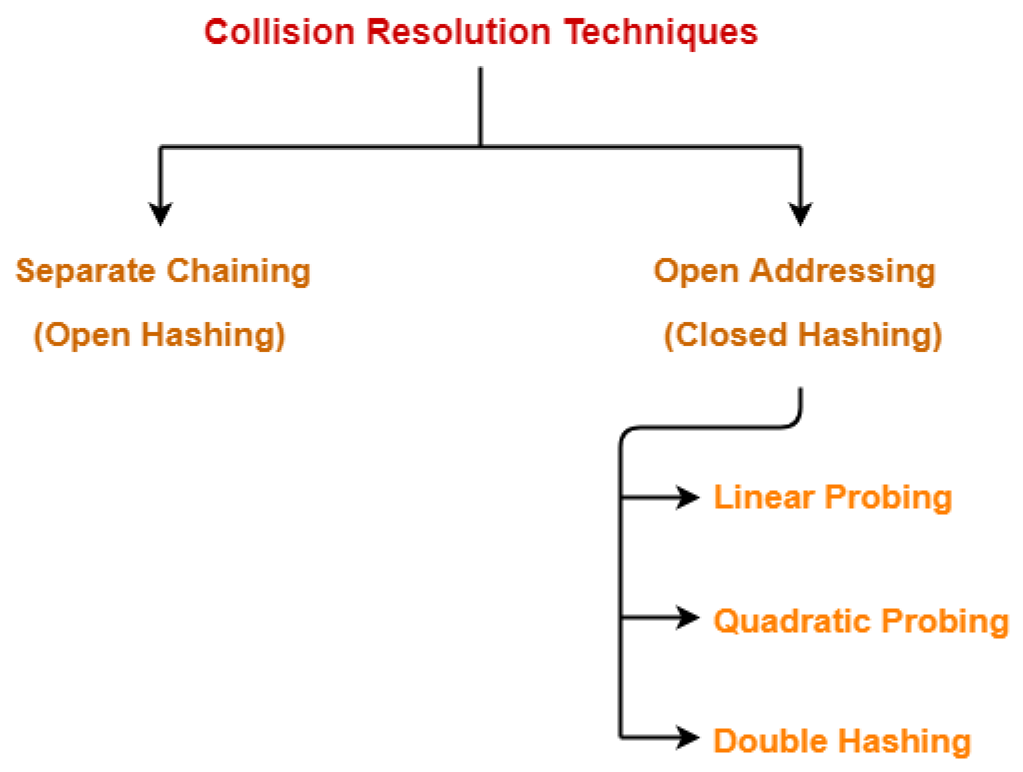
- Hash function is used to compute the hash value for a key.
- Hash value is then used as an index to store the key in the hash table.
- Hash function may return the same hash value for two or more keys.

When the hash value of a key maps to an already occupied bucket of the hash table, it is called as a **Collision**.

Collision Resolution Techniques-

Collision Resolution Techniques are the techniques used for resolving or handling the collision.

Collision resolution techniques are classified as-



1. Separate Chaining
2. Open Addressing

Separate Chaining-

To handle the collision,

- This technique creates a linked list to the slot for which collision occurs.
- The new key is then inserted in the linked list.
- These linked lists to the slots appear like chains.
- That is why, this technique is called as **separate chaining**.

Time Complexity-

For Searching-

- In worst case, all the keys might map to the same bucket of the hash table.
- In such a case, all the keys will be present in a single linked list.
- Sequential search will have to be performed on the linked list to perform the search.
- So, time taken for searching in worst case is O(n).

For Deletion-

- In worst case, the key might have to be searched first and then deleted.
- In worst case, time taken for searching is O(n).
- So, time taken for deletion in worst case is O(n).

Load Factor (α)-

Load factor (α) is defined as-

$$\text{Load Factor } (\alpha) = \frac{\text{Number of elements present in the hash table}}{\text{Total size of the hash table}}$$

If Load factor (α) = constant, then time complexity of Insert, Search, Delete = Θ(1)

PRACTICE PROBLEM BASED ON SEPARATE CHAINING-

Problem-

Using the hash function 'key mod 7', insert the following sequence of keys in the hash table-

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

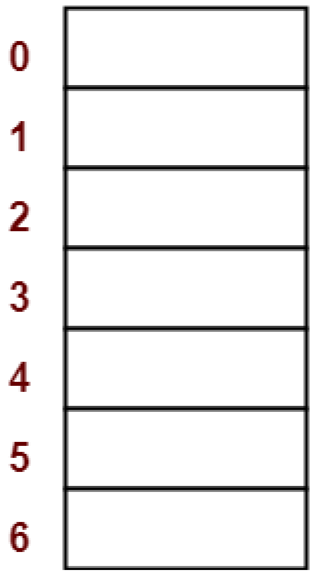
Use separate chaining technique for collision resolution.

Solution-

The given sequence of keys will be inserted in the hash table as-

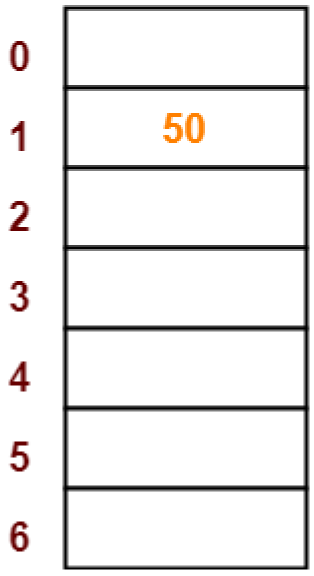
Step-01:

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-



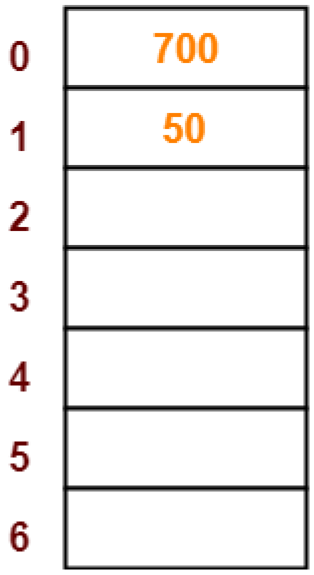
Step-02:

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-



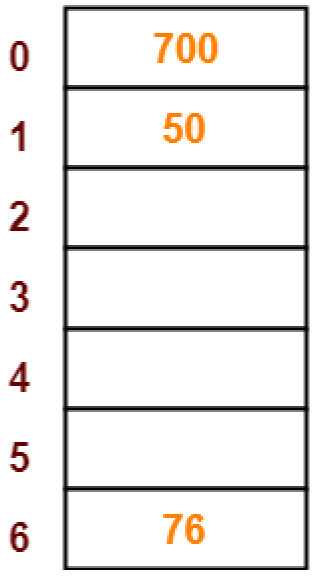
Step-03:

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



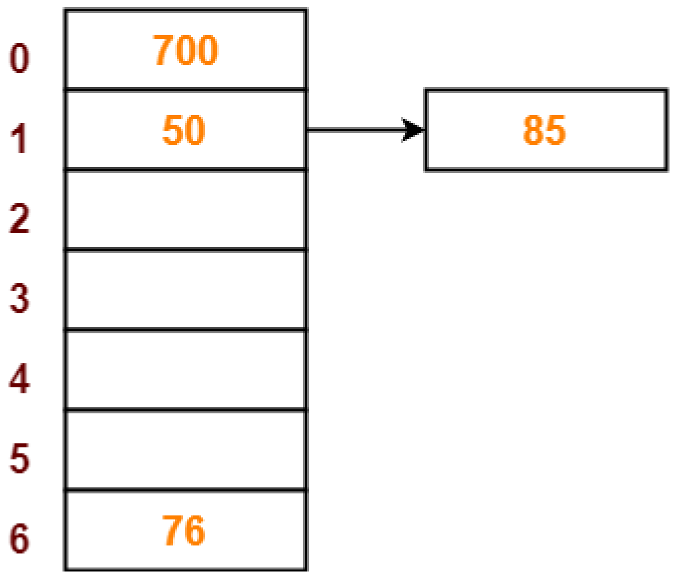
Step-04:

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-



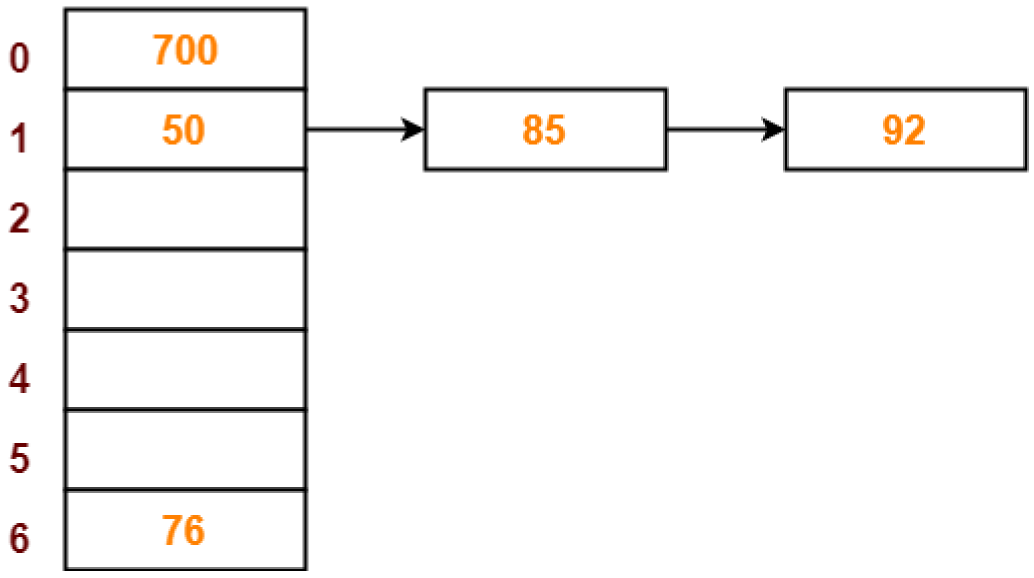
Step-05:

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 85 will be inserted in bucket-1 of the hash table as-



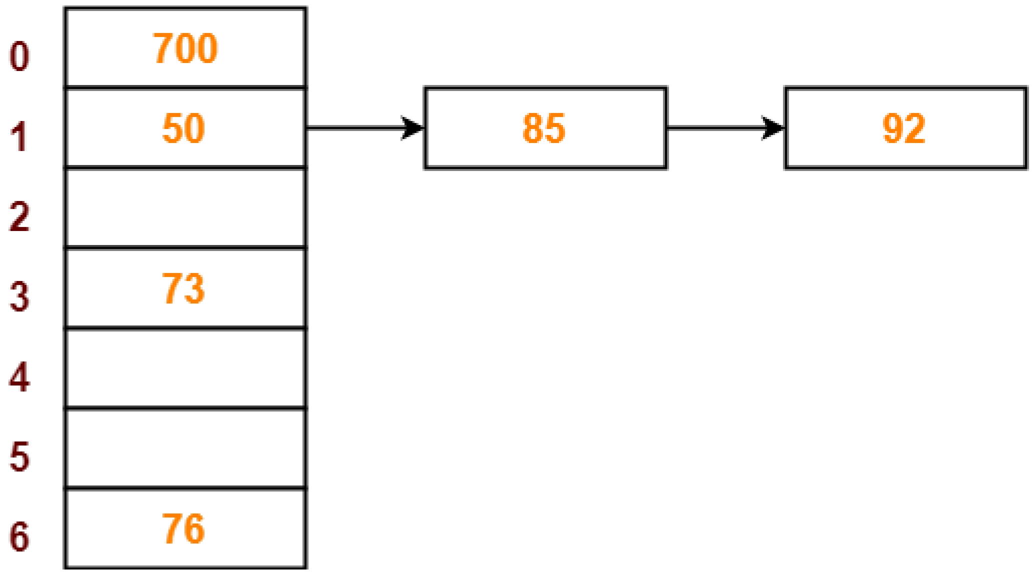
Step-06:

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 92 will be inserted in bucket-1 of the hash table as-



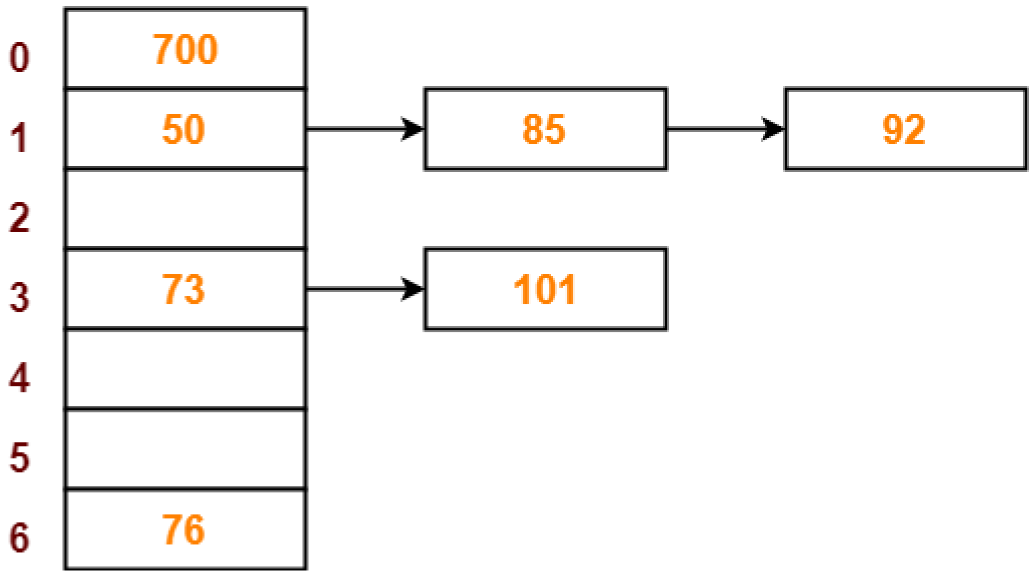
Step-07:

- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
- So, key 73 will be inserted in bucket-3 of the hash table as-



Step-08:

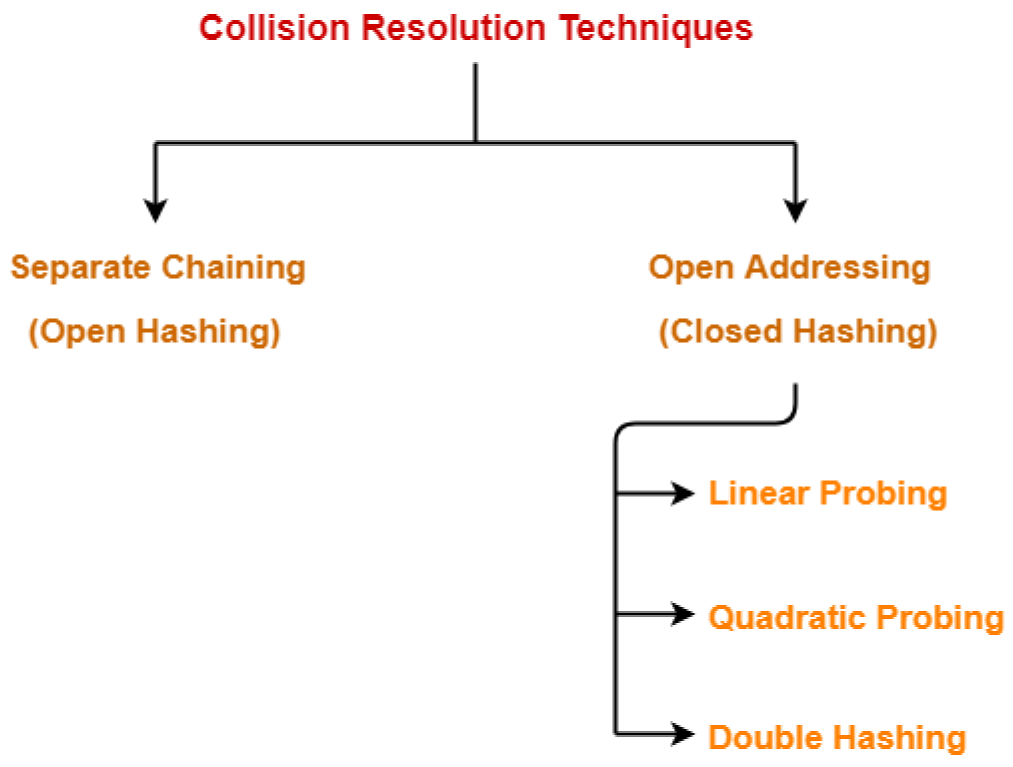
- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-3.
- So, key 101 will be inserted in bucket-3 of the hash table as-



Collision Resolution Techniques-

We have discussed-

- **Hashing** is a well-known searching technique.
- Collision occurs when hash value of the new key maps to an occupied bucket of the hash table.
- Collision resolution techniques are classified as-



Open Addressing-

In open addressing,

- Unlike separate chaining, all the keys are stored inside the hash table.
- No key is stored outside the hash table.

Techniques used for open addressing are-

- Linear Probing
- Quadratic Probing
- Double Hashing

Operations in Open Addressing-

Let us discuss how operations are performed in open addressing-

Insert Operation-

- Hash function is used to compute the hash value for a key to be inserted.
- Hash value is then used as an index to store the key in the hash table.

In case of collision,

- Probing is performed until an empty bucket is found.
- Once an empty bucket is found, the key is inserted.
- Probing is performed in accordance with the technique used for open addressing.

Search Operation-

To search any particular key,

- Its hash value is obtained using the hash function used.
- Using the hash value, that bucket of the hash table is checked.
- If the required key is found, the key is searched.
- Otherwise, the subsequent buckets are checked until the required key or an empty bucket is found.
- The empty bucket indicates that the key is not present in the hash table.

Delete Operation-

- The key is first searched and then deleted.
- After deleting the key, that particular bucket is marked as “deleted”.

NOTE-

- During insertion, the buckets marked as “deleted” are treated like any other empty bucket.
- During searching, the search is not terminated on encountering the bucket marked as “deleted”.
- The search terminates only after the required key or an empty bucket is found.

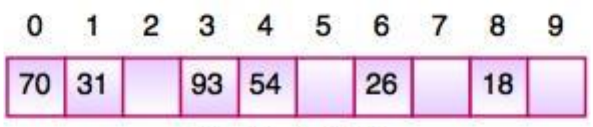
Open Addressing Techniques-

Techniques used for open addressing are-

- Take the above example, if we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem. This problem is called as **Collision** or **Clash**. Collision creates a problem for hashing technique.
- **Linear probing is used for resolving the collisions in hash table**, data structures for maintaining a collection of key-value pairs.
- Linear probing was invented by Gene Amdahl, Elaine M. McGraw and Arthur Samuel in 1954 and analyzed by Donald Knuth in 1963.
- It is a component of open addressing scheme for using a hash table to solve the dictionary problem.
- The simplest method is called Linear Probing. Formula to compute linear probing is:

P = (1 + P) % (MOD) Table_size

For example,



0	1	2	3	4	5	6	7	8	9
70	31		93	54		26		18	

Fig. Hash Table

If we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem.

Linear probing solves this problem:

P = H(40)
44 % 10 = **0**
Position 0 is occupied by 70. so we look elsewhere for a position to store 40.

Using Linear Probing:
P= (P + 1) % table-size
0 + 1 % 10 = **1**
But, position 1 is occupied by 31, so we look elsewhere for a position to store 40.

Using linear probing, we try next position : 1 + 1 % 10 = 2
Position 2 is empty, so 40 is inserted there.

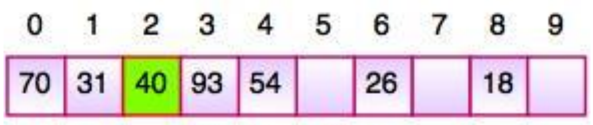


Fig. Hash Table

In linear probing,

- When collision occurs, we linearly probe for the next bucket.
- We keep probing until an empty bucket is found.

Advantage-

- It is easy to compute.

Disadvantage-

- The main problem with linear probing is clustering.
- Many consecutive elements form groups.
- Then, it takes time to search an element or to find an empty bucket.

Time Complexity-

Worst time to search an element in linear probing is O (table size).

This is because-

- Even if there is only one element present and all other elements are deleted.
- Then, “deleted” markers present in the hash table makes search the entire table.

Problem-

Using the hash function ‘key mod 7’, insert the following sequence of keys in the hash table-

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

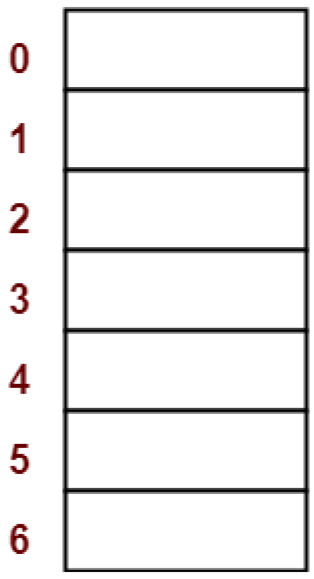
Use linear probing technique for collision resolution.

Solution-

The given sequence of keys will be inserted in the hash table as-

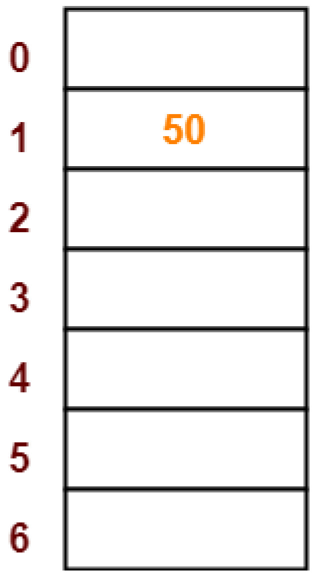
Step-01:

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-



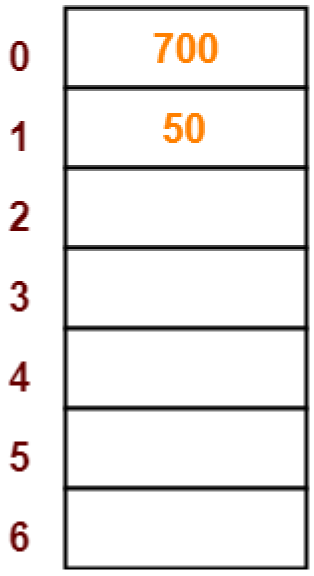
Step-02:

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-



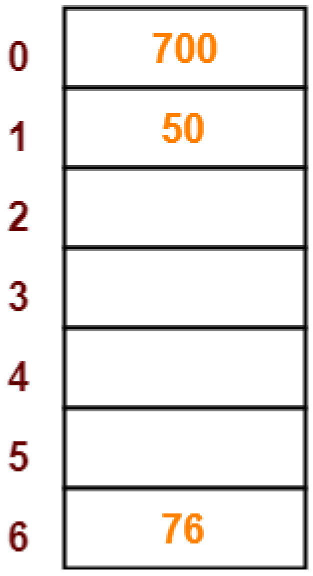
Step-03:

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



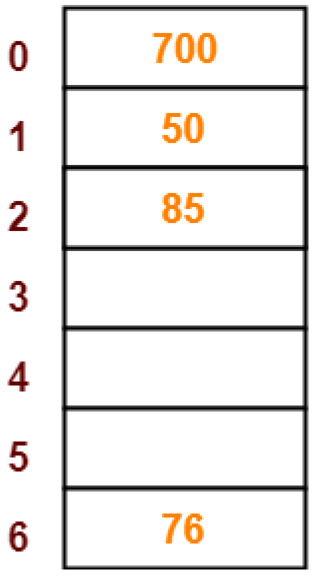
Step-04:

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-



Step-05:

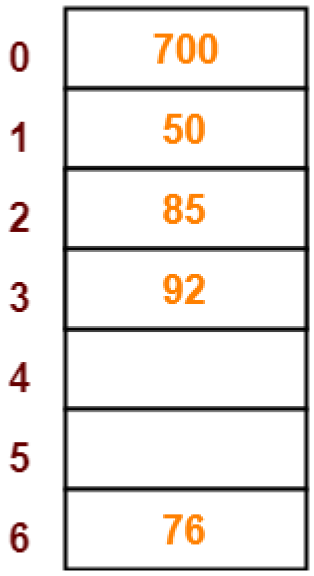
- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-2.
- So, key 85 will be inserted in bucket-2 of the hash table as-



Step-06:

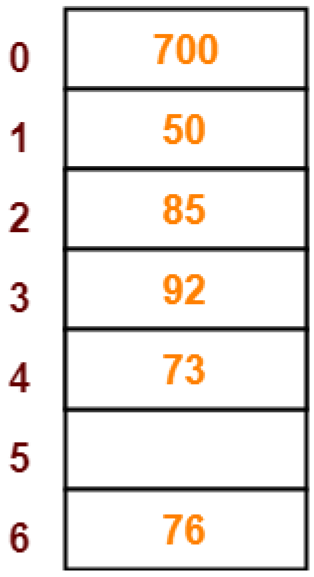
- The next key to be inserted in the hash table = 92.

- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-3.
- So, key 92 will be inserted in bucket-3 of the hash table as-



Step-07:

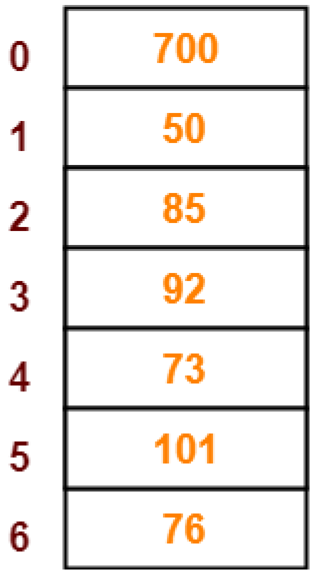
- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-4.
- So, key 73 will be inserted in bucket-4 of the hash table as-



0	700
1	50
2	85
3	92
4	73
5	
6	76

Step-08:

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-5.
- So, key 101 will be inserted in bucket-5 of the hash table as-



0	700
1	50
2	85
3	92
4	73
5	101
6	76

2. Quadratic Probing-

In quadratic probing,

- When collision occurs, we probe for i^2‘th bucket in i^{th} iteration.
- We keep probing until an empty bucket is found.

3. Double Hashing-

In double hashing,

- We use another hash function hash2(x) and look for i * hash2(x) bucket in i^{th} iteration.
- It requires more computation time as two hash functions need to be computed.

Comparison of Open Addressing Techniques-

	Linear Probing	Quadratic Probing	Double Hashing
Primary Clustering	Yes	No	No
Secondary Clustering	Yes	Yes	No
Number of Probe Sequence (m = size of table)	m	m	m^2
Cache performance	Best	Lies between the two	Poor

Conclusions-

- Linear Probing has the best cache performance but suffers from clustering.
- Quadratic probing lies between the two in terms of cache performance and clustering.
- Double caching has poor cache performance but no clustering.

Load Factor (α)-

Load factor (α) is defined as-

$$\boxed{\text{Load Factor } (\alpha) = \frac{\text{Number of elements present in the hash table}}{\text{Total size of the hash table}}}$$

In open addressing, the value of load factor always lie between 0 and 1.

This is because-

- In open addressing, all the keys are stored inside the hash table.
- So, size of the table is always greater or at least equal to the number of keys stored in the table.

Double hashing is a collision resolving technique in **Open Addressed** Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

Double hashing can be done using :

(hash1(key) + i * hash2(key)) % TABLE_SIZE
Here hash1() and hash2() are hash functions and TABLE_SIZE
is size of hash table.
(We repeat by increasing i when collision occurs)

First hash function is typically hash1(key) = key % TABLE_SIZE

A popular second hash function is :

hash2(key) = PRIME – (key % PRIME)

where PRIME is a prime smaller than the TABLE_SIZE.
A good second Hash function is:

- It must never evaluate to zero
- Must make sure that all cells can be probed

Lets say, Hash1 (key) = key % 13

Hash2 (key) = 7 – (key % 7)

Hash1(19) = 19 % 13 = 6

Hash1(27) = 27 % 13 = 1

Hash1(36) = 36 % 13 = 10

Hash1(10) = 10 % 13 = 10

Hash2(10) = 7 – (10%7) = 4

(Hash1(10) + 1*Hash2(10))%13= 1

(Hash1(10) + 2*Hash2(10))%13= 5

Collision