Insertion Sort Algorithm

1. This is an in-place comparison-based sorting algorithm.
2. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sorts**.
3. The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).
4. This algorithm is not suitable for large data sets as its average and worst case complexity are of **$O(n^2)$,** where **n** is the number of items.

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

Pseudocode

```
Procedure insertionSort( A : array of items )
   int holePosition
   int valueToInsert

   for i = 1 to length(A) inclusive do:

      /* select value to be inserted */
      valueToInsert = A[i]
      holePosition = i

      /*locate hole position for the element to be inserted */

      while holePosition > 0 and A[holePosition-1] > valueToInsert do:
         A[holePosition] = A[holePosition-1]
         holePosition = holePosition -1
      end while

      /* insert the number at hole position */
      A[holePosition] = valueToInsert

   end for

end procedure
```

Example - How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list.

Selection Sort Algorithm

1. Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
2. The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.
3. Limitation- This algorithm is not suitable for large data sets as its average and worst case complexities are of **$O(n^2)$** where n is the number of items.

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

Pseudocode

```
procedure selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for

end procedure
```

Example- How Selection Sort Works?

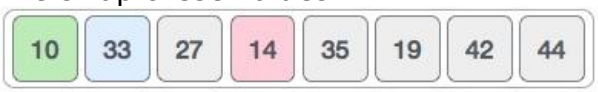
Consider the following depicted array as an example.

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

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

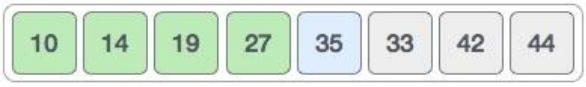
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

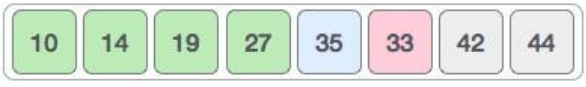
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

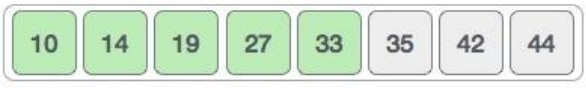


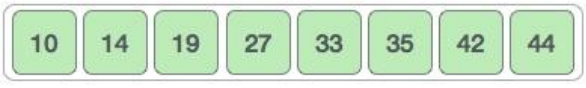
After two iterations, two least values are positioned at the beginning in a sorted manner.

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









Merge sort Algorithm

1. Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being **O(n log n),** it is one of the most respected algorithms.
2. Merge sort first divides the array into equal halves and then combines them in a sorted manner.
3. Merge sort keeps on dividing the list into equal parts until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

```
Algorithm
Step 1 – if it is only one element in the list it is already sorted, return.
```

```
Step 2 – divide the list recursively into two parts until it can no more be divided.
Step 3 – merge the smaller lists into new list in sorted order.
```

```
Merge Sort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

Pseudocode

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions – divide & merge.

Merge sort works with recursion and our implementation in the same way.

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a
   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]
   l1 = mergesort( l1 )
   l2 = mergesort( l2 )
   return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

   var c as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while

   while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a
   end while

   while ( b has elements )
      add b[0] to the end of c
      remove b[0] from b
   end while

   return c
end procedure
```

Example- How **Merge Sort** Works?

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

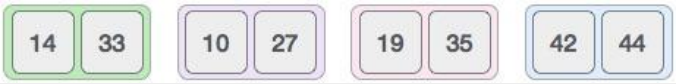
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

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

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

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

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



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

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

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

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

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is .

Time complexity of Merge Sort is in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Merge Sort Algorithm

Merge Sort is a kind of Divide and Conquer algorithm in computer programrming. It is one of the most popular sorting algorithms and a great way to develop confidence in building recursive algorithms.

Divide and Conquer Strategy

Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array `A`. A subproblem would be to sort a sub-section of this array starting at index `p` and ending at index `r`, denoted as `A[p..r]`.

Divide

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r]

The MergeSort Algorithm
The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. p == r.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

```
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)
```

To sort an entire array, we need to call MergeSort(A, 0, length(A)-1).
As shown in the image below, the merge sort algorithm recursively divides the array into halves until we reach the base case of array with 1 element. After that, the merge function picks up the sorted sub-arrays and merges them to gradually sort the entire array.

The merge step of merge sort
Every recursive algorithm is dependent on a base case and the ability to combine the results from base cases. Merge sort is no different. The most important part of the merge sort algorithm is, you guessed it, the "merge" step.
The merge step is the solution to the simple problem of merging two sorted lists(arrays) to build one large sorted list(array).
The algorithm maintains three pointers, one for each of the two arrays and one for maintaining the current index of final sorted array.

```
Have we reached the end of any of the arrays?
    No:
        Compare current elements of both arrays
        Copy smaller element into sorted array
        Move pointer of element containing smaller element
    Yes:Copy all remaining elements of non-empty array
```

Writing the code for merge algorithm
A noticeable difference between the merging step we described above and the one we use for merge sort is that we only perform the merge function on consecutive sub-arrays.
This is why we only need the array, the first position, the last index of the first subarray(we can calculate the first index of second subarray) and the last index of second subarray.
Our task is to merge two subarrays `A[p..q]` and `A[q+1..r]` to create a sorted array `A[p..r]`. So the inputs to the function are A, p, q and r
The merge function works as follows:

1. Create copies of the subarrays L ← `A[p..q]` and `M ← A[q+1..r]`.
2. Create three pointers `i`,`j` and k
    1. `i` maintains current index of `L`, starting at 1

2. `j` maintains current index of `M`, starting at 1
3. `k` maintains current index of `A[p..q]`, starting at `p`

3. Until we reach the end of either `L` or `M`, pick the larger among the elements from `L` and `M` and place them in the correct position at `A[p..q]`
4. When we run out of elements in either `L` or `M`, pick up the remaining elements and put in `A[p..q]`

In code, this would look like:

```
void merge(int A[], int p, int q, int r)
{
    /* Create L ← A[p..q] and M ← A[q+1..r] */
    int n1 = q - p + 1;
    int n2 =  r - q;

    int L[n1], M[n2];

    for (i = 0; i < n1; i++)
        L[i] = A[p + i];
    for (j = 0; j < n2; j++)
        M[j] = A[q + 1 + j];

    /* Maintain current index of sub-arrays and main array */
    int i, j, k;
    i = 0;
    j = 0;
    k = p;

    /* Until we reach either end of either L or M, pick larger among elements L and M
and place them in the correct position at A[p..r] */
    while (i < n1 && j < n2)
    {
        if (L[i] <= M[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = M[j];
            j++;
        }
        k++;
    }
     /* When we run out of elements in either L or M, pick up the remaining elements and
put in A[p..r] */
    while (i < n1)
    {
        A[k] = L[i];
        i++;
        k++;
    }

    while (j < n2)
    {
        A[k] = M[j];
        j++;
        k++;
```

```
    }
}
```

Merge function explained step-by-step

There is a lot happening in this function, so let's take an example to see how this would work.
As usual, a picture speaks a thousand words.

The array `A[0..8]` contains two sorted subarrays `A[1..5]` and `A[6..7]`. Let us see how merge function will merge the two arrays.

```
void merge(int A[], int p = 1, int q = 4, int 6)
{
```

Step 1: Create duplicate copies of sub-arrays to be sorted

```
    /* Create L ← A[p..q] and M ← A[q+1..r] */
    n1 = 4 - 1 + 1 = 4;
    n2 =  6 - 4 = 2;

    int L[4], M[2];

    for (i = 0; i < 4; i++)
        L[i] = A[p + i];
    /* L[0,1,2,3] = A[1,2,3,4] = [1,5,10,12] */

    for (j = 0; j < 2; j++)
        M[j] = A[q + 1 + j];
    /* M[0,1,2,3] = A[5,6] = [6,9]
```

Step 2: Maintain current index of sub-arrays and main array

```
    int i, j, k;
    i = 0;
    j = 0;
    k = p;
```

Step 3: Until we reach end of either L or M, pick larger among elements L and M and place them in the correct position at A[p..r]

```
    while (i < n1 && j < n2) {
        if (L[i] <= M[j]) {
            A[k] = L[i]; i++;
        }
        else {
            A[k] = M[j];
            j++;
       }
        k++;
    }
```

Step 4: When we run out of elements in either L or M, pick up the remaining elements and put in A[p..r]

```
/* We exited the earlier loop because j < n2 doesn't hold */
    while (i < n1)
    {
        A[k] = L[i];
        i++;
        k++;
    }
```

```
/* We exited the earlier loop because i < n1 doesn't hold */

    while (j < n2)
    {
        A[k] = M[j];
        j++;
        k++;
    }
}
```

This step would have been needed if size of M was greater than L. At the end of the merge function, the subarray `A[p..r]` is sorted.

Quick Sort Algorithm

Quick sort is a fast sorting algorithm used to sort a list of elements. Quick sort algorithm is invented by **C. A. R. Hoare**. The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it use **divide and conquer** strategy. In quick sort, the partition of the list is performed based on the element called ***pivot***. Here pivot element is one of the elements in the list.
The list is divided into two partitions such that **"all elements to the left of pivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot"**.

Step by Step Process

In Quick sort algorithm, partitioning of the list is performed using following steps...

- **Step 1 -** Consider the first element of the list as **pivot** (i.e., Element at first position in the list).
- **Step 2 -** Define two variables i and j. Set i and j to first and last elements of the list respectively.
- **Step 3 -** Increment i until list[i] > pivot then stop.
- **Step 4 -** Decrement j until list[j] < pivot then stop.
- **Step 5 -** If i < j then exchange list[i] and list[j].
- **Step 6 -** Repeat steps 3,4 & 5 until i > j.
- **Step 7 -** Exchange the pivot element with list[j] element.

Following is the sample code for Quick sort...

```
//Quick Sort Logic

void quickSort(int list[10],int first,int last){
    int pivot,i,j,temp;
```

```
 if(first < last){
     pivot = first;
     i = first;
     j = last;

     while(i < j){
         while(list[i] <= list[pivot] && i < last)
             i++;
         while(list[j] && list[pivot])
             j--;
         if(i < j){
              temp = list[i];
              list[i] = list[j];
              list[j] = temp;
         }
     }

     temp = list[pivot];
     list[pivot] = list[j];
     list[j] = temp;
     quickSort(list,first,j-1);
     quickSort(list,j+1,last);
}
```

Consider the following unsorted list of elements...

Define pivot, left & right. Set pivot = 0, left = 1 & right = 7. Here '7' indicates 'size-1'.

Compare List[left] with List[pivot]. If **List[left]** is greater than **List[pivot]** then stop left otherwise move left to the next.
Compare List[right] with List[pivot]. If List[right] is smaller than List[pivot] then stop right otherwise move right to the previous.
Repeat the same until **left>=right**.
If both left & right are stoped but left<right then swap List[left] with List[right] and countinue the process.
If left>=right then swap List[pivot] with List[right].

Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 8.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 2.

Here left & right both are stoped and left is not greater than right so we need to swap List[left] and List[right]

Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 6.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 4.

Here left & right both are stoped and left is greater than right so we need to swap List[pivot] and List[right]

Here we can observe that all the numbers to the left side of 5 are smaller and right side are greater. That means 5 is placed in its correct position.
Repeat the same process on the left sublist and right sublist to the number 5.

In the left sublist as there are no smaller number than the pivot left will keep on moving to the next and stops at last number. As the List[right] is smaller, right stops at same position. Now left and right both are equal so we swap pivot with right.

In the right sublist left is grester than the pivot, left will stop at same position.
As the List[right] is greater than List[pivot], right moves towords left and stops at pivot number position.
Now left > right so we swap pivot with right. (6 is swap by itself).

Repeat the same recursively on both left and right sublists until all the numbers are sorted.
The final sorted list will be as follows...

}

Heap Sort Algorithm

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

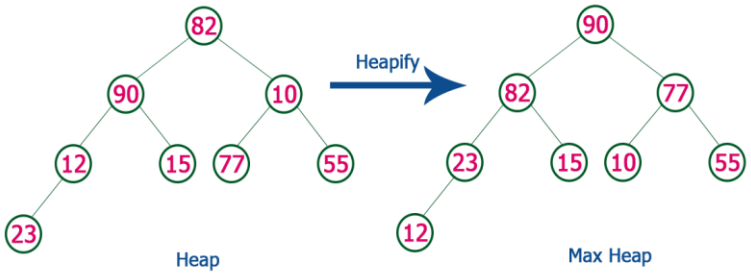
The Heap sort algorithm to arrange list of elements in ascending order is performed using following steps...

- **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

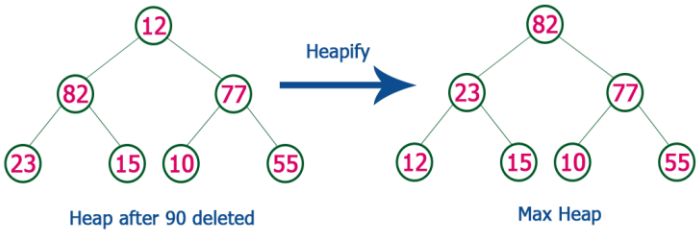
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

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

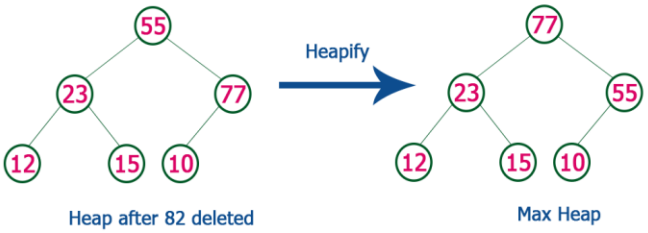
Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



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.

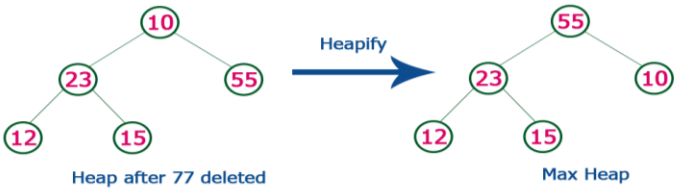


list of numbers after swapping 82 with 55.

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.

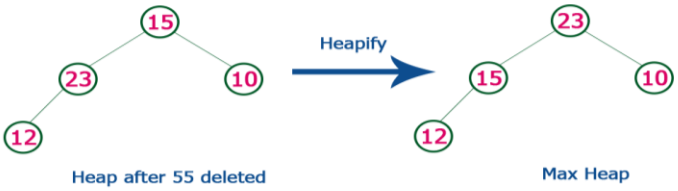
with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 77 with 10.

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

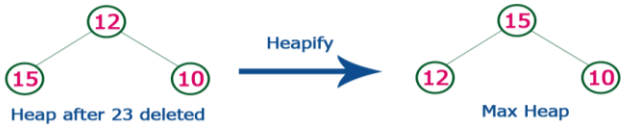
Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

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

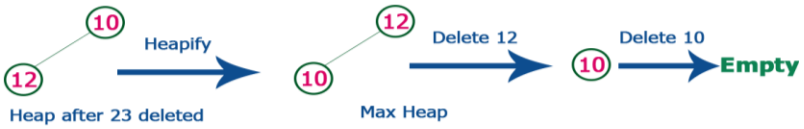
Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



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

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

Static Hashing

In all search techniques like linear search, binary search and search trees, the time required to search an element depends on the total number of elements present in that data structure. In all these search techniques, as the number of elements increases the time required to search an element also increases linearly.

Hashing is an approach in which time required to search an element doesn't depends on the total number of elements. Using hashing data structure, a given element is searched with **constant time complexity**. Hashing is an effective way to reduce the number of comparisons to search an element in a data structure.

Hashing is defined as follows...

Hashing is the process of indexing and retrieving element (data) in a data structure to provide faster way of finding the element using hash key.

Here, hash key is a value which provides the index value where the actual data is likely to be stored in the data structure.

In this data structure, we use a concept called **Hash table** to store data. All the data values are inserted into the hash table based on the hash key value. Hash key value is used to map the data with index in the hash table. And the hash key is generated for every data using a **hash function**. That means every entry in the hash table is based on the hash key value generated using hash function.

Hash Table is defined as follows...

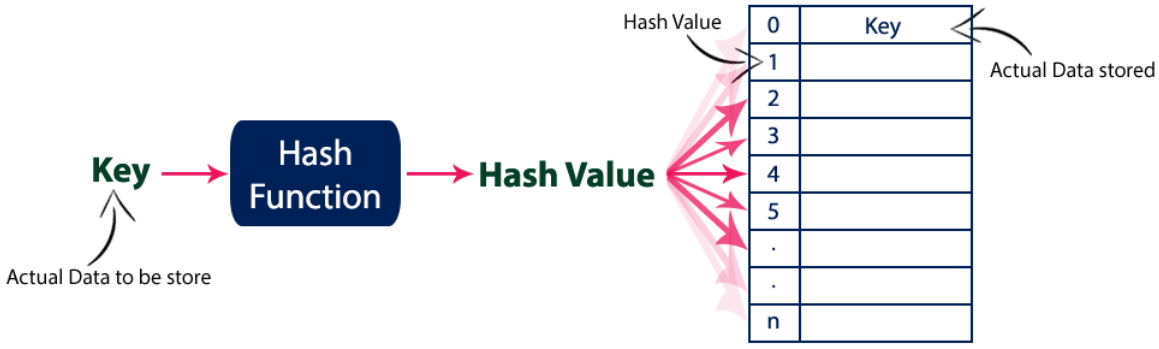
Hash table is just an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations are performed with constant time complexity (i.e. O(1)).

Hash tables are used to perform insertion, deletion and search operations very quickly in a data structure. Using hash table concept, insertion, deletion and search operations are accomplished in constant time complexity. Generally, every hash table makes use of a function called **hash function** to map the data into the hash table.

A hash function is defined as follows...

Hash function is a function which takes a piece of data (i.e. key) as input and produces an integer (i.e. hash value) as output which maps the data to a particular index in the hash table.

Basic concept of hashing and hash table is shown in the following figure...



What is Sparse Matrix?

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represents a m X n matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

Sparse matrix is a matrix which contains very few non-zero elements.

When a sparse matrix is represented with 2-dimensional array, we waste lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times. To make it simple we use the following sparse matrix representation.

Sparse Matrix Representations

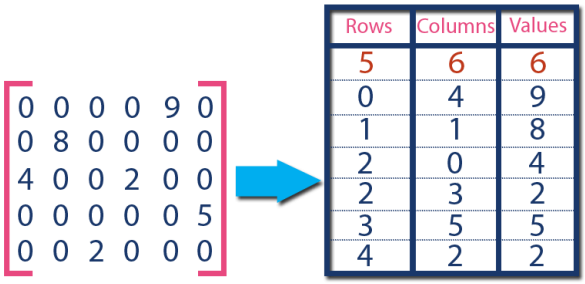
A sparse matrix can be represented by using TWO representations, those are as follows...

1. Triplet Representation (Array Representation)
2. Linked Representation

Triplet Representation (Array Representation)

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0th row stores total number of rows, total number of columns and total number of non-zero values in the sparse matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...



Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. Second row is filled with 0, 4, & 9 which indicates the non-zero value 9 is at 0th row 4th column in the Sparse matrix. In the same way the remaining non-zero values also follows the similar pattern.

Implementation of Array Representation of Sparse Matrix using C++

```
#include<iostream>
using namespace std;
int main()
{
   // sparse matrix of class 5x6 with 6 non-zero values
   int sparseMatrix[5][6] =
   {
      {0 , 0 , 0 , 0 , 9, 0 },
      {0 , 8 , 0 , 0 , 0, 0 },
      {4 , 0 , 0 , 2 , 0, 0 },
      {0 , 0 , 0 , 0 , 0, 5 },
      {0 , 0 , 2 , 0 , 0, 0 }
   };
   // Finding total non-zero values in the sparse matrix
   int size = 0;
   for (int row = 0; row < 5; row++)
      for (int column = 0; column < 6; column++)
         if (sparseMatrix[row][column] != 0)
            size++;
   // Defining result Matrix
   int resultMatrix[3][size];
   // Generating result matrix
   int k = 0;
   for (int row = 0; row < 5; row++)
      for (int column = 0; column < 6; column++)
         if (sparseMatrix[row][column] != 0)
         {
            resultMatrix[0][k] = row;
            resultMatrix[1][k] = column;
            resultMatrix[2][k] = sparseMatrix[row][column];
            k++;
         }
   // Displaying result matrix
   cout<<"Triplet Representation : "<<endl;
   for (int row=0; row<3; row++)
   {
      for (int column = 0; column<size; column++)
         cout<<resultMatrix[row][column]<<" ";
      cout<<endl;
   }
return 0;
}
```