SORTING

Sorting refers to arranging data in a particular order (ascending or descending order).

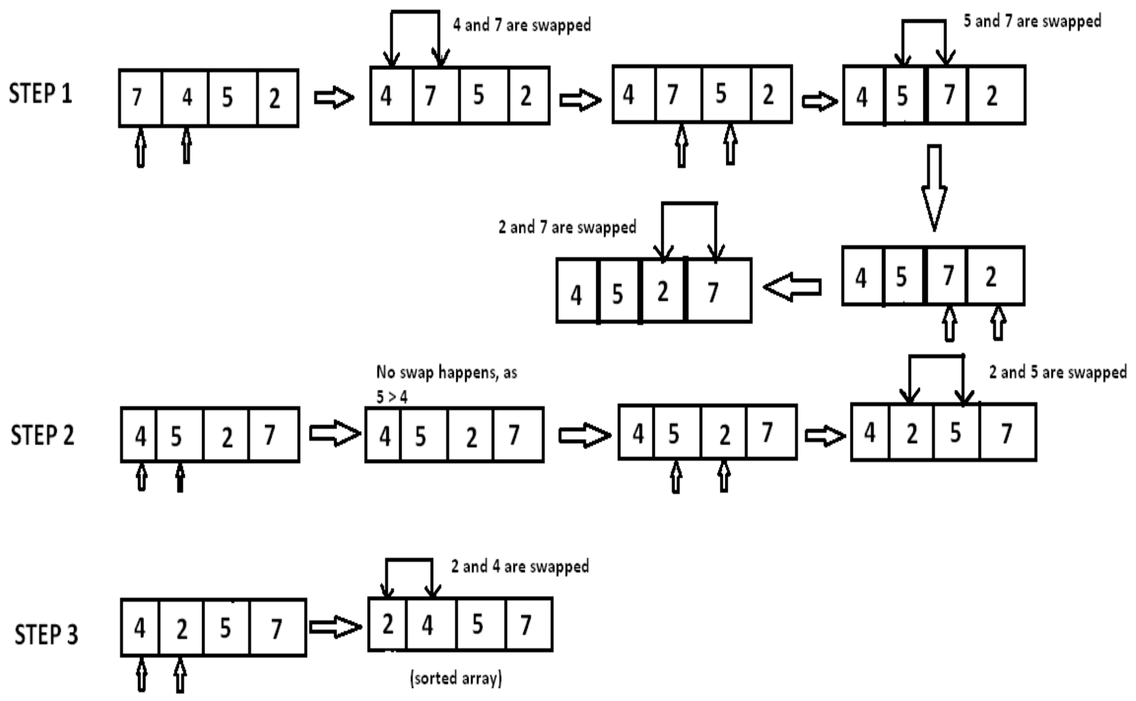
Types of Sorting :

1. Bubble sort
2. Selection sort
3. Insertion sort
4. Quick sort
5. Merge sort

1.Bubble sort

Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist in the wrong order.

Lets try to understand with an example: A [ ] = { 7, 4, 5, 2}



In step 1, 7 is compared with 4. Since 7>4, 7 is moved ahead of 4. Since all the other elements are of a lesser value than 7, 7 is moved to the end of the array.

Now the array is A[]={4,5,2,7}.

In step 2, 4 is compared with 5. Since 5>4 and both 4 and 5 are in ascending order, these elements are not swapped. However, when 5 is compared with 2, 5>2 and these elements are in descending order. Therefore, 5and 2 are swapped.

Now the array is A[]={4,2,5,7}.

In step 3, the element 4 is compared with 2. Since 4>2 and the elements are in descending order, 4 and 2 are swapped.

The sorted array is A[]={2,4,5,7}.

2.Selection sort

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with remaining all the elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped. Then we select the element at second position in the list and it is compared with remaining all elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated till the entire list is sorted.

The selection sort algorithm is performed using following steps...

Step 1: Select the first element of the list (i.e., Element at first position in the list).

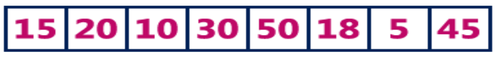
Step 2: Compare the selected element with all other elements in the list.

Step 3: For every comparison, if any element is smaller than selected element (for Ascending order), then these two are swapped.

Step 4: Repeat the same procedure with next position in the list till the entire list is sorted.

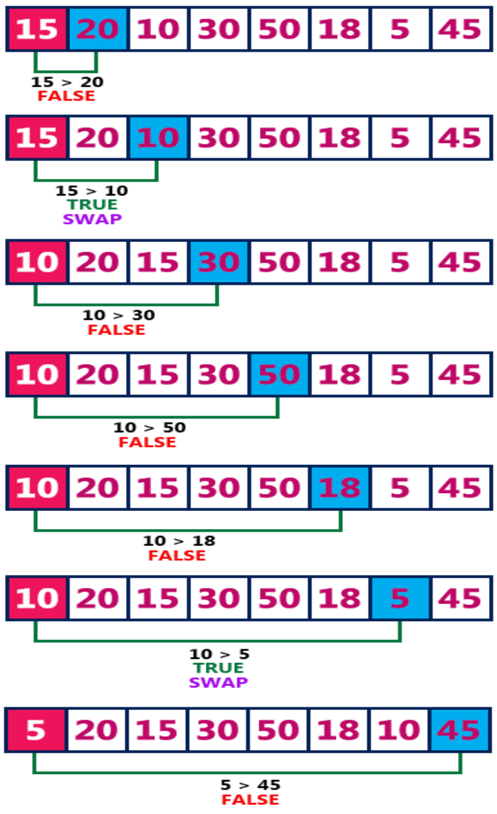
Example of selection sort

Consider the following unsorted list of elements...

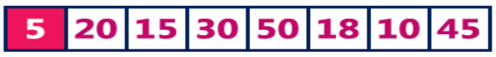


Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.



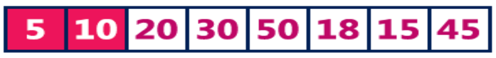
List after 1st iteration



Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

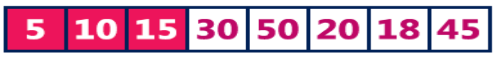
List after 2nd iteration



Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

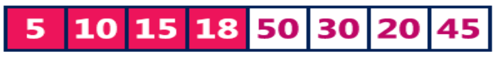
List after 3rd iteration



Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

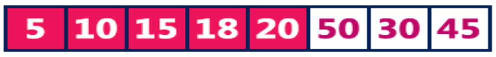
List after 4th iteration



Iteration #5

Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

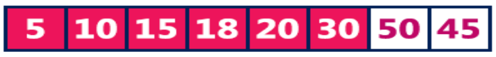
List after 5th iteration



Iteration #6

Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

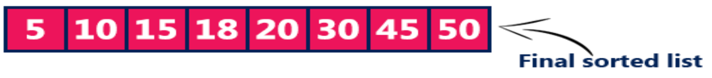
List after 6th iteration



Iteration #7

Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 7th iteration



3.Insertion Sort

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

The insertion sort algorithm is performed using following steps...

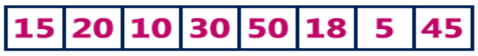
Step 1: Asume that first element in the list is in sorted portion of the list and remaining all elements are in unsorted portion.

Step 2: Consider first element from the unsorted list and insert that element into the sorted list in order specified.

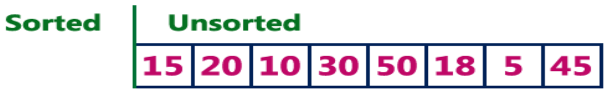
Step 3: Repeat the above process until all the elements from the unsorted list are moved into the sorted list.

Example of Insertion sort

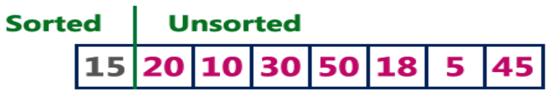
Consider the following unsorted list of elements...



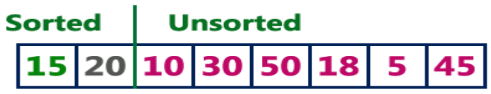
Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...



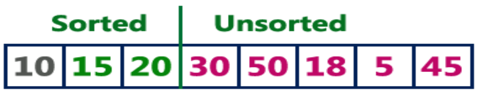
Move the first element 15 from unsorted portion to sorted portion of the list.



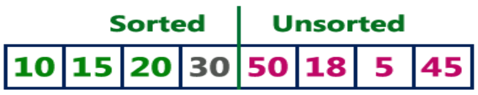
To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position



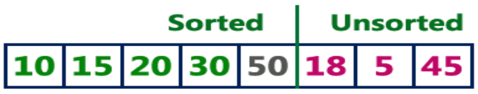
To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.



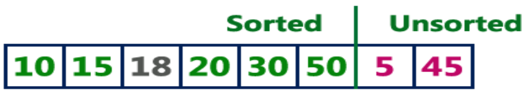
To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.



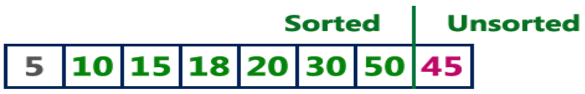
To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.



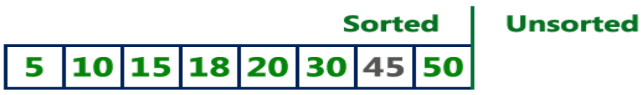
To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.



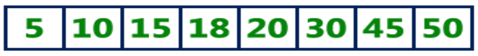
To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.



To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.



Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...

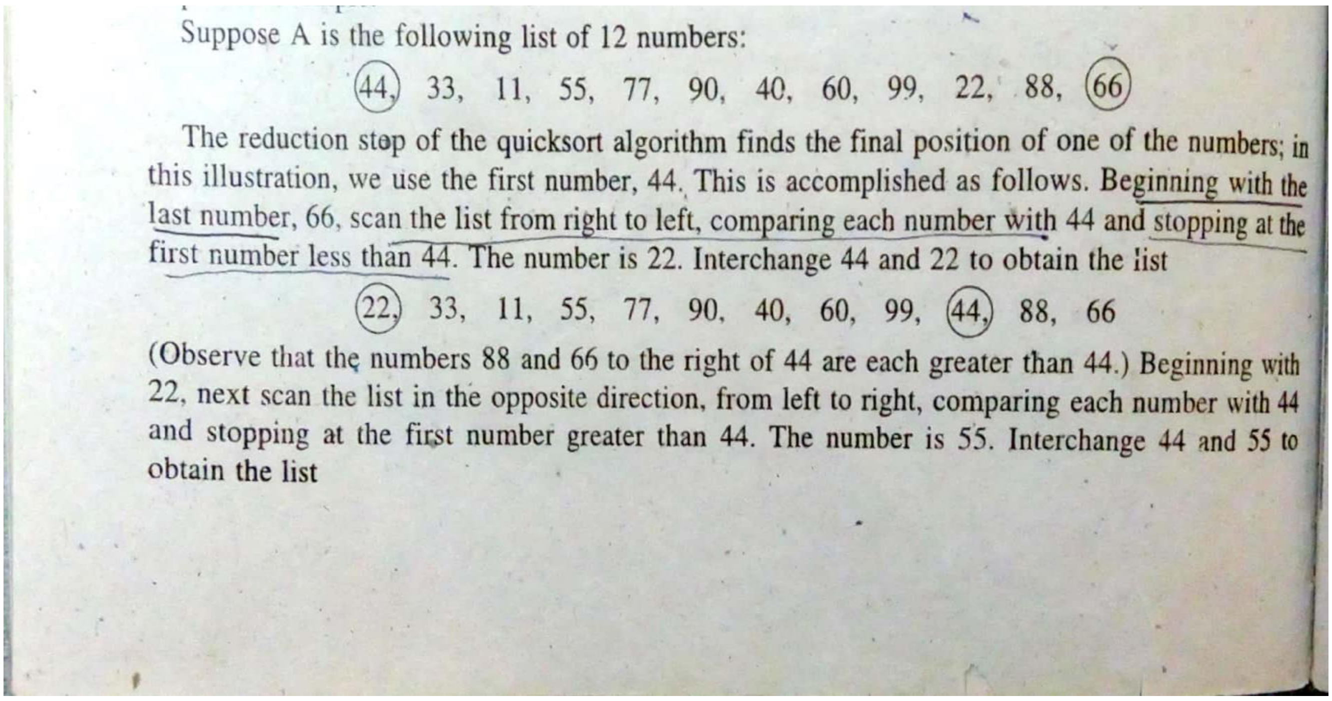


4.Quick sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value. Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quick Sort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

Example of quick sort



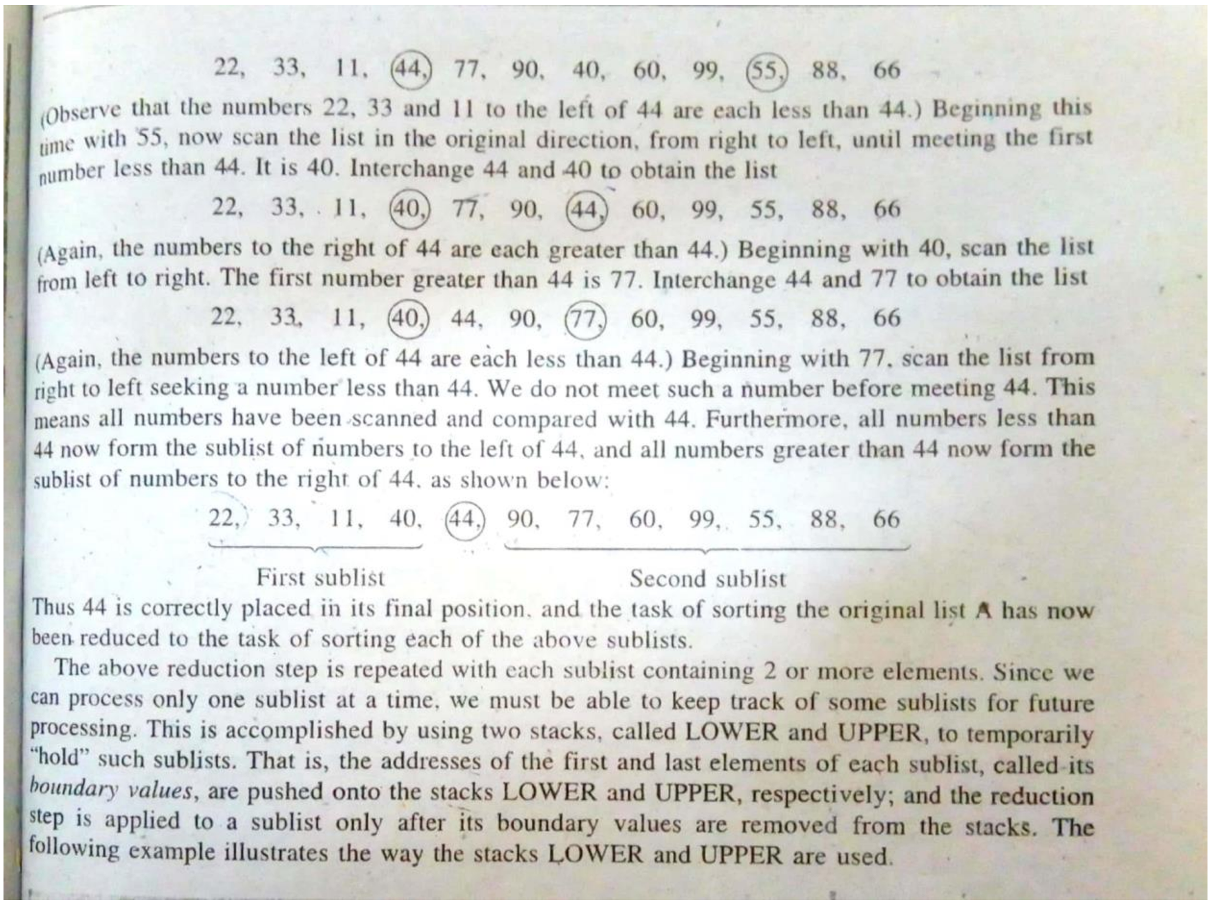
Suppose A is the following list of 12 numbers:

$\boxed{44}$, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, $\boxed{66}$

The reduction step of the quicksort algorithm finds the final position of one of the numbers; in this illustration, we use the first number, 44. This is accomplished as follows. Beginning with the last number, 66, scan the list from right to left, comparing each number with 44 and stopping at the first number less than 44. The number is 22. Interchange 44 and 22 to obtain the list

$\boxed{22}$, 33, 11, 55, 77, 90, 40, 60, 99, $\boxed{44}$, 88, 66

(Observe that the numbers 88 and 66 to the right of 44 are each greater than 44.) Beginning with 22, next scan the list in the opposite direction, from left to right, comparing each number with 44 and stopping at the first number greater than 44. The number is 55. Interchange 44 and 55 to obtain the list



22, 33, 11, 44, 77, 90, 40, 60, 99, 55, 88, 66

(Observe that the numbers 22, 33 and 11 to the left of 44 are each less than 44.) Beginning this time with 55, now scan the list in the original direction, from right to left, until meeting the first number less than 44. It is 40. Interchange 44 and 40 to obtain the list

22, 33, 11, 40, 77, 90, 44, 60, 99, 55, 88, 66

(Again, the numbers to the right of 44 are each greater than 44.) Beginning with 40, scan the list from left to right. The first number greater than 44 is 77. Interchange 44 and 77 to obtain the list

22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 88, 66

(Again, the numbers to the left of 44 are each less than 44.) Beginning with 77, scan the list from right to left seeking a number less than 44. We do not meet such a number before meeting 44. This means all numbers have been scanned and compared with 44. Furthermore, all numbers less than 44 now form the sublist of numbers to the left of 44, and all numbers greater than 44 now form the sublist of numbers to the right of 44, as shown below:

$$\underbrace{22,\ 33,\ 11,\ 40,}_{\text{First sublist}}\ 44,\ \underbrace{90,\ 77,\ 60,\ 99,\ 55,\ 88,\ 66}_{\text{Second sublist}}$$

Thus 44 is correctly placed in its final position, and the task of sorting the original list A has now been reduced to the task of sorting each of the above sublists.

The above reduction step is repeated with each sublist containing 2 or more elements. Since we can process only one sublist at a time, we must be able to keep track of some sublists for future processing. This is accomplished by using two stacks, called LOWER and UPPER, to temporarily "hold" such sublists. That is, the addresses of the first and last elements of each sublist, called its *boundary values*, are pushed onto the stacks LOWER and UPPER, respectively; and the reduction step is applied to a sublist only after its boundary values are removed from the stacks. The following example illustrates the way the stacks LOWER and UPPER are used.

<u>NOTE –</u> For all the sorting programs refer the txt files.

Q. WAP in C to implement selection sort.

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

void main()
{
    int a[5]={4,1,9,3,6};
    int i,j,min,temp;


    printf("\nElements before sorting\n ");

    for(i=0;i<5;i++)
    {
        printf("%d ",a[i]);
    }

    for(i=0;i<4;i++)
    {
        min=i;
        for(j=i+1;j<5;j++)
        {
            if(a[min]>a[j])
            min=j;
        }
        if(min!=i)
        {
            temp=a[i];
            a[i]=a[min];
            a[min]=temp;
        }
    }

    printf("\nElements after sorting\n");
    for(i=0;i<5;i++)
    {
        printf("%d ",a[i]);
    }
    getch();
}
```

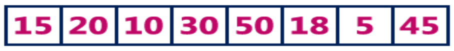
Q. WAP in C to implement bubble sort.

```
#include<stdio.h>

int main()
{
    int a[5]= {40,10,20,70,30};
    int i,j,temp;
    printf("\nArray elements before sorting: \n");
    for(i=0;i<5;++i)
        printf("%d ",a[i]);

    for(i=1;i<5;++i)
        for(j=0;j<(5-i);++j)
            if(a[j]>a[j+1])
            {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
            }

    printf("\nArray elements after sorting: \n");
    for(i=0;i<5;++i)
        printf("%d ",a[i]);

    return 0;
}
```

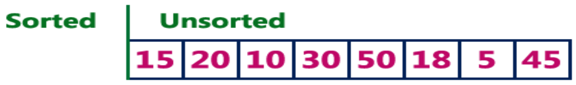
Insertion Sort

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

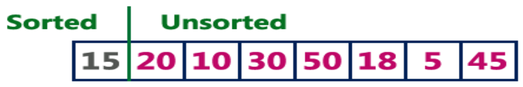
Consider the following unsorted list of elements...



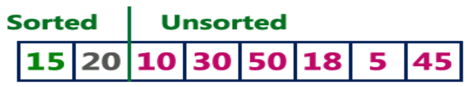
Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...



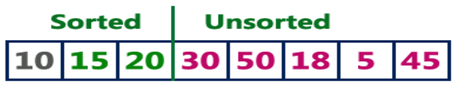
Move the first element 15 from unsorted portion to sorted portion of the list.



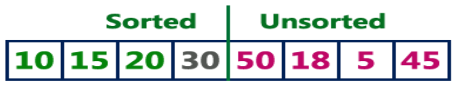
To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position



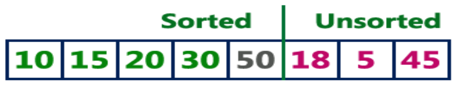
To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.



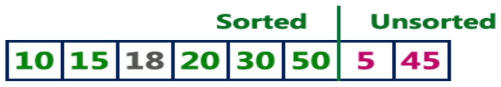
To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.



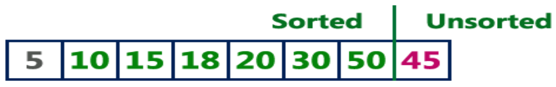
To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.



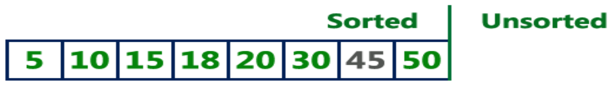
To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.



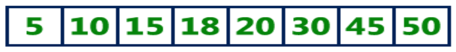
To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.



To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.



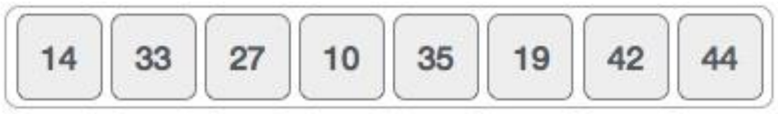
Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...



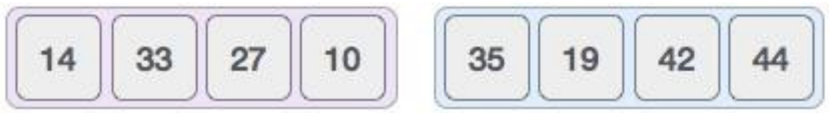
Complexity of the Insertion Sort Algorithm
Worst Case : O(n^2)
Best Case : Ω(n)
Average Case : Θ(n^2)

Merge sort

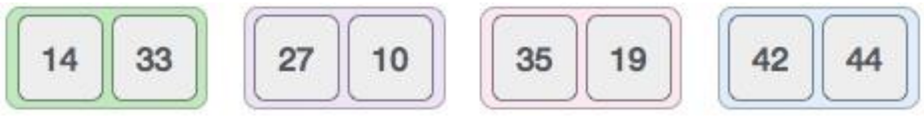
Merge sort is a sorting technique based on divide and conquer technique. Merge sort first divides the array into equal halves and then combines them in a sorted manner. Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.



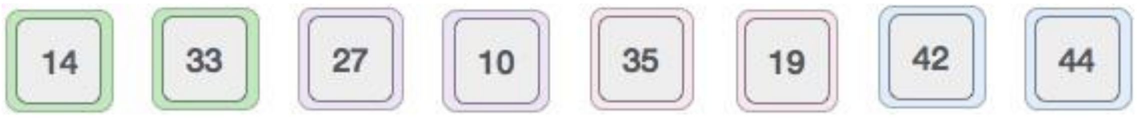
We see here that an array of 8 items is divided into two arrays of size 4.



Now we divide these two arrays into halves.

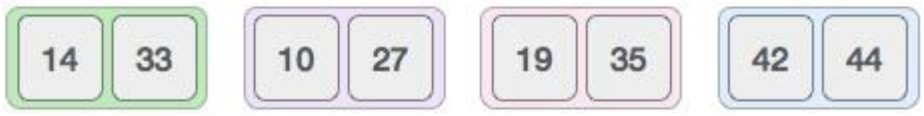


We further divide these arrays

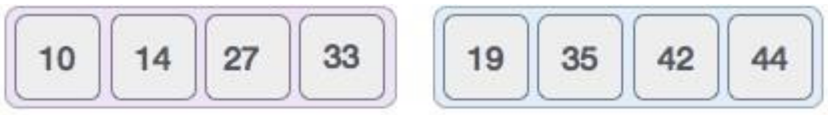


Now, we combine them in exactly the same manner as they were broken down.

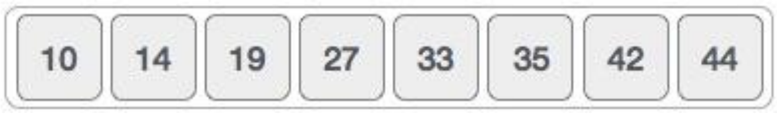
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed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

C-Program

Q. WAP in C to implement insertion sort.

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

void main()
{
  int a[5]= {20,5,70,30,10};
  int i, j, temp;

  printf("\nArray elements before Sorting :\n ");
  for (i = 0; i < 5; i++)
  printf(" %d", a[i]);



  for (i = 1; i < 5; 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;
    }

    printf("\nArray elements after Sorting : \n");
    for (i = 0; i < 5; i++)
      printf(" %d", a[i]);
   getch();
}
```

Q. WAP in C to implement merge sort.

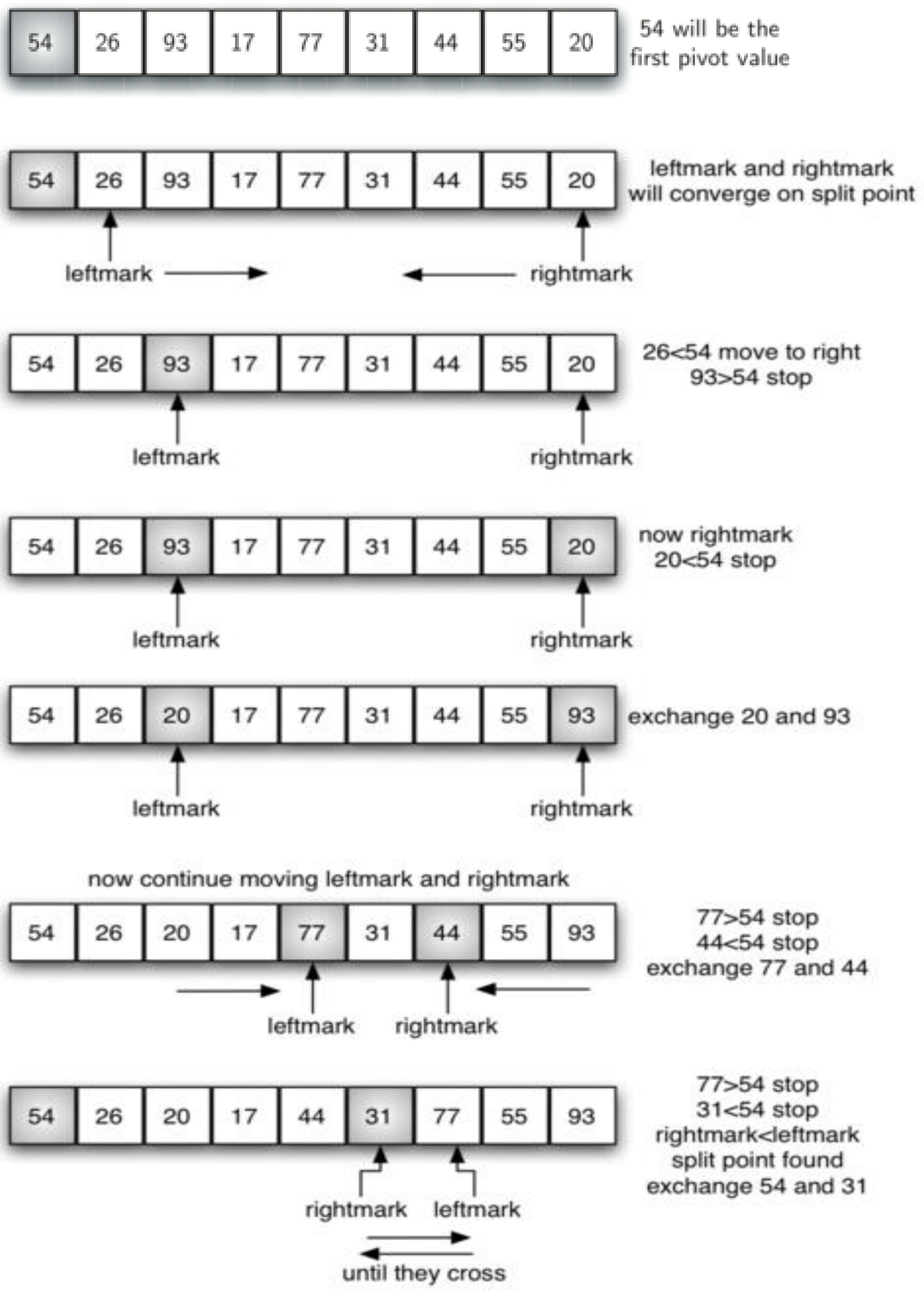
```
#include<stdio.h>
#include<conio.h>

void main()
{
    int i;
    int a[]={11,66,88,33,66,77,99,88,22,44,55};
    printf("\nArray elements before sorting\n");
    for(i=0;i<11;i++)
    {
        printf("%d",a[i]);
    }
        mergesort(a,11);
        printf("\n");
    printf("\nArray elements after sorting\n");
    for(i=0;i<11;i++)
    {
        printf("%d",a[i]);
    }
    getch();
}

void mergesort(int A[],int N)
{
    int L =1,B[11];
    while(L<N)
    {
        mergepass(A,N,L,B);
        mergepass(B,N,2*L,A);
        L=4*L;
    }
}

void mergepass(int A[],int N,int L,int B[])
{
    int j,LB;
    int Q,S,R;
    Q=N/(2*L);
    S=2*L*Q;
    R=N-S;

    for(j=0;j<Q;j++)
    {
        LB=(2*j)*L;
        merge(A,L,LB,A,L,LB+L,B,LB);
    }
    if(R<=L)
    {
        for(j=0;j<R;j++)
        B[S+j]=A[S+j];
    }
    else
    {
        merge(A,L,S,A,R-L,S+L,B,S);
    }
}

void merge(int A[],int n1,int index1,int B[],int n2,int index2,int C[],int index)
{
    while(n1&&n2)
    {
        if(A[index1]<B[index2])
        {
            C[index]=A[index1];
            index++;
            index1++;
            n1--;
        }
        else
        {
            C[index]=B[index2];
            index++;
            index2++;
            n2--;

        }
    }
    while(n1)
    {
        C[index]=A[index1];
        index++;
        index1++;
        n1--;
    }
    while(n2)
    {
        C[index]=B[index2];
        index++;
        index2++;
        n2--;
    }
}
```

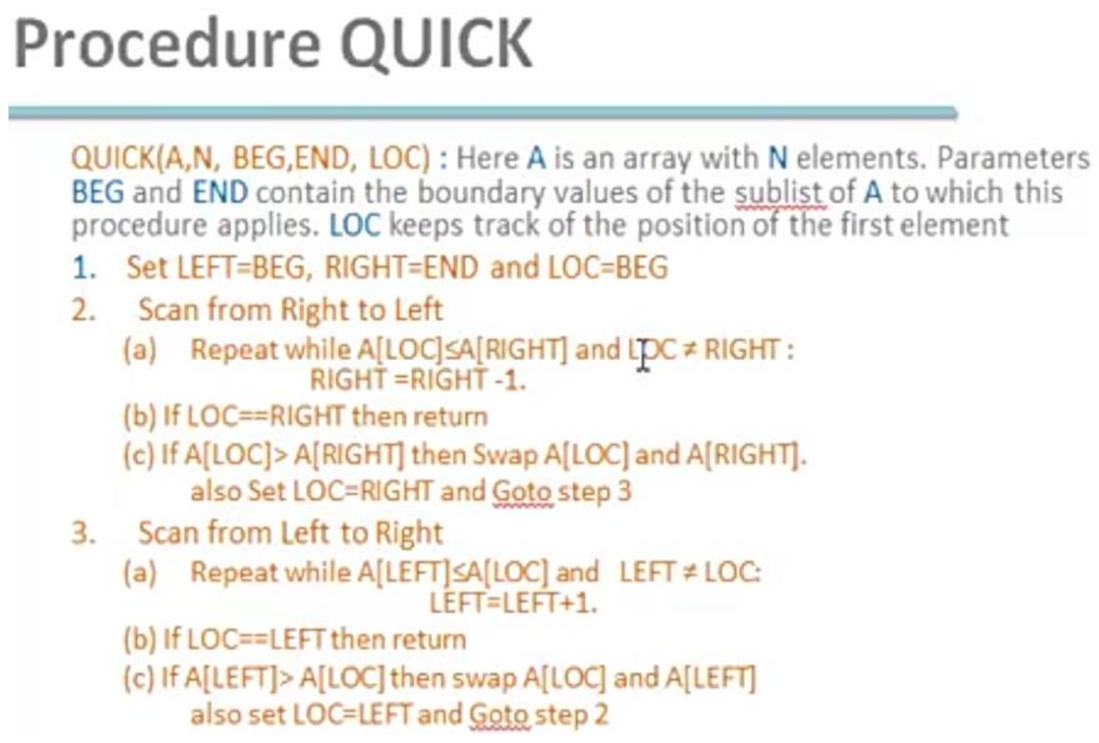
Quick sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value. QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

Example





Procedure QUICK

QUICK(A,N, BEG,END, LOC) : Here A is an array with N elements. Parameters BEG and END contain the boundary values of the sublist of A to which this procedure applies. LOC keeps track of the position of the first element

```
1. Set LEFT=BEG, RIGHT=END and LOC=BEG
2. Scan from Right to Left
   (a) Repeat while A[LOC]≤A[RIGHT] and LOC ≠ RIGHT :
               RIGHT =RIGHT -1.
   (b) If LOC==RIGHT then return
   (c) If A[LOC]> A[RIGHT] then Swap A[LOC] and A[RIGHT].
       also Set LOC=RIGHT and Goto step 3
3. Scan from Left to Right
   (a) Repeat while A[LEFT]≤A[LOC] and  LEFT ≠ LOC:
               LEFT=LEFT+1.
   (b) If LOC==LEFT then return
   (c) If A[LEFT]> A[LOC] then swap A[LOC] and A[LEFT]
       also set LOC=LEFT and Goto step 2
```

Complexity of the Quick Sort Algorithm

Worst Case : **$O(n^2)$**
Best Case : **O (n log n)**
Average Case : **O (n log n)**

Q. WAP in C to implement quick sort.

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

void quicksort(int [10],int,int);

void main()
{
  int a[5]={20,5,70,30,10};
  int i;

  printf("\nElements before sorting:\n ");

  for(i = 0; i < 5; i++)
  printf(" %d",a[i]);

  quicksort(a,0,5-1);

  printf("\nElements after sorting :\n ");
  for(i = 0; i < 5; i++)
    printf(" %d",a[i]);

  getch();
}

void quicksort(int a[10],int first,int last)
{
    int pivot,i,j,temp;

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

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

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

```
          a[j] = temp;
          quicksort(a,first,j-1);
          quicksort(a,j+1,last);

     }
}
```

Searching

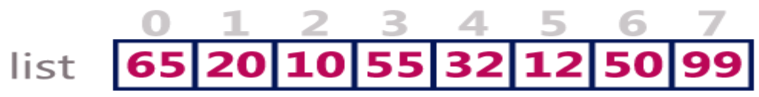
Searching is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Types of searching techniques:

1. **Linear Search Algorithm (Sequential Search Algorithm)**

Linear search algorithm finds a given element in a list of elements with **O(n)** time complexity where **n** is total number of elements in the list. This search process starts comparing search element with the first element in the list. If both are matched then result is element found otherwise search element is compared with next element in the list. Repeat the same until search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

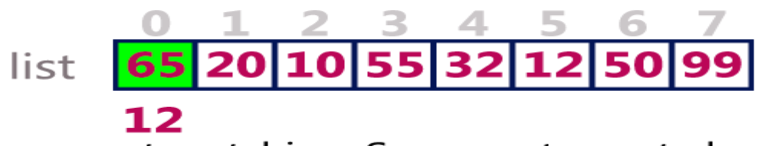
Example of linear search



search element 12

Step 1:

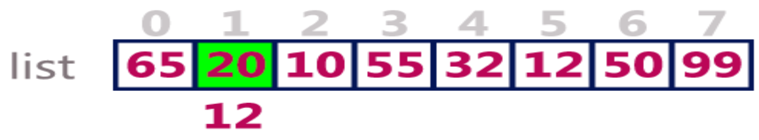
search element (12) is compared with first element (65)



Both are not matching. So move to next element

Step 2:

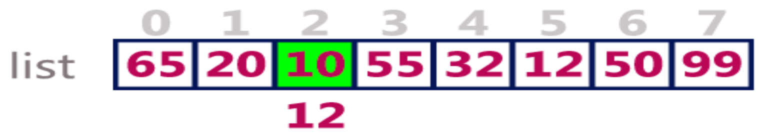
search element (12) is compared with next element (20)



Both are not matching. So move to next element

Step 3:

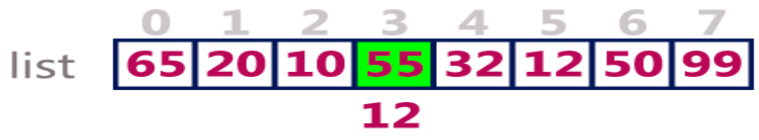
search element (12) is compared with next element (10)



Both are not matching. So move to next element

Step 4:

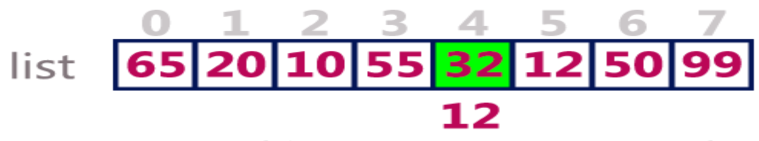
search element (12) is compared with next element (55)



Both are not matching. So move to next element

Step 5:

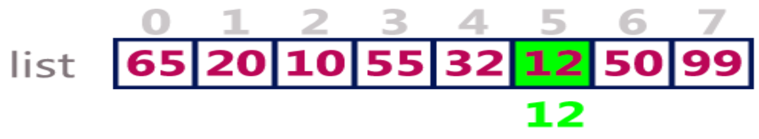
search element (12) is compared with next element (32)



Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

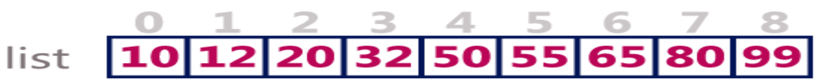


Both are matching. So we stop comparing and display element found at index 5.

2. Binary Search Algorithm

Binary search algorithm finds a given element in a list of elements with **O (log n)** time complexity where **n** is total number of elements in the list. The binary search algorithm can be used with only sorted list of elements. That means, binary search is used only with list of elements that are already arranged in an order. The binary search cannot be used for list of elements arranged in random order. This search process starts comparing the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sub list of the middle element. If the search element is larger, then we repeat the same process for right sub list of the middle element. We repeat this process until we find the search element in the list or until we left with a sub list of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

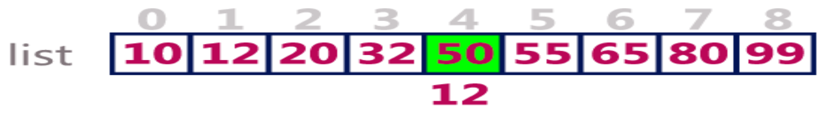
Example of binary search



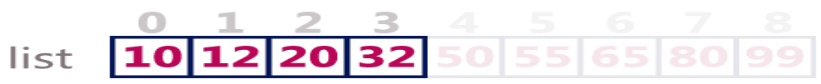
search element **12**

Step 1:

search element (12) is compared with middle element (50)

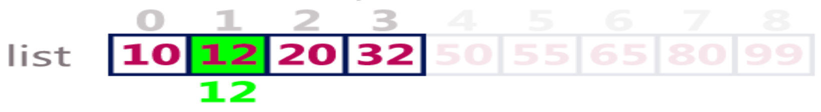


Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).



Step 2:

search element (12) is compared with middle element (12)

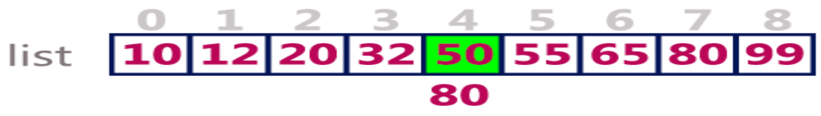


Both are matching. So the result is "Element found at index 1"

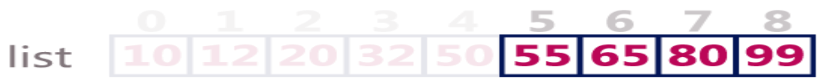
search element **80**

Step 1:

search element (80) is compared with middle element (50)

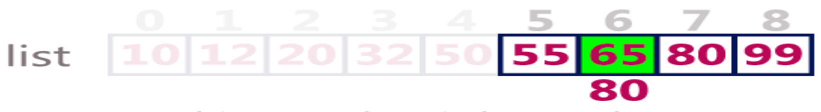


Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

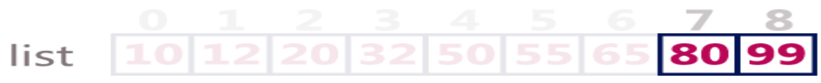


Step 2:

search element (80) is compared with middle element (65)

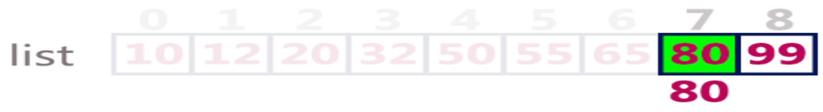


Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).



Step 3:

search element (80) is compared with middle element (80)



Both are not matching. So the result is "Element found at index 7"

3. Hashing

Hashing is another approach in which time required to search an element doesn't depend 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:

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

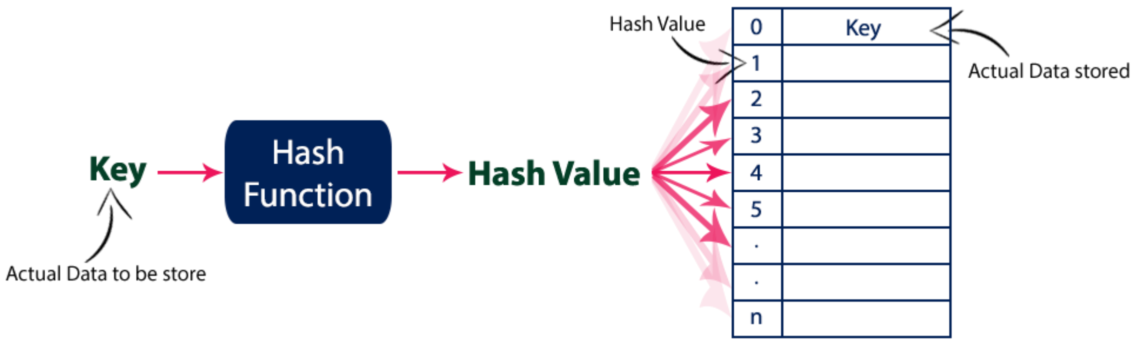
Hash table is just an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations are performed with constant time complexity (i.e. O(1)).

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:

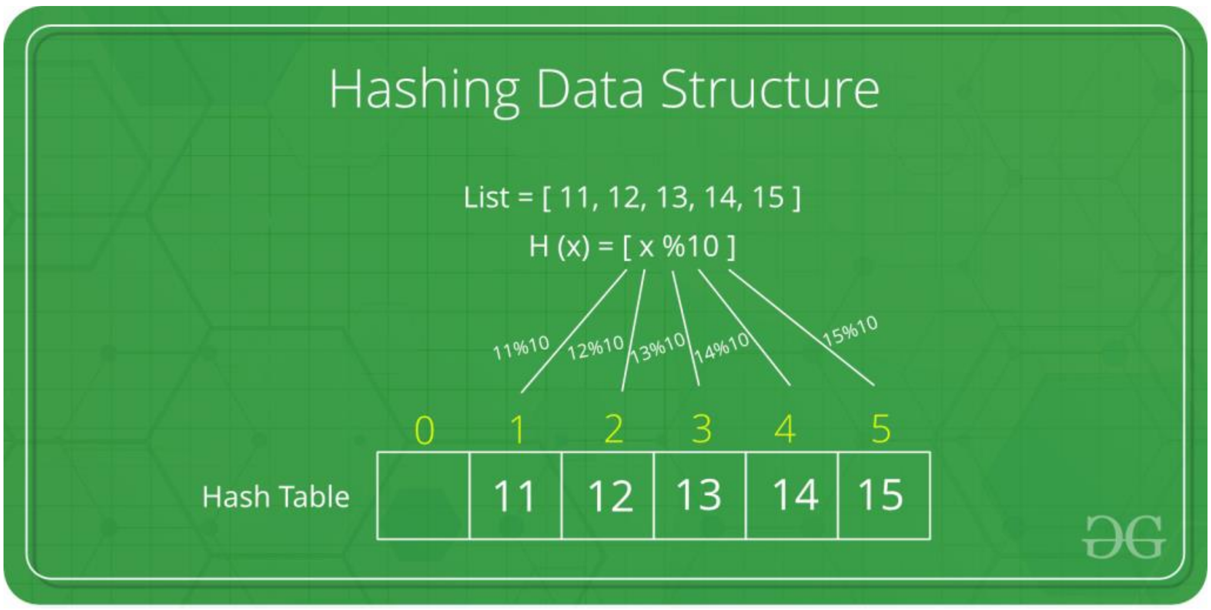
Hash function is a function which takes a piece of data (i.e. key) as input and produces an integer (i.e. hash value) as output which maps the data to a particular index in the hash table.

Basic concept of hashing and hash table is shown in this diagram



Example

Assume a hash function H(x) maps the value at the index **x%10** in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



➢ <u>**Collision in Hashing-**</u>

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.

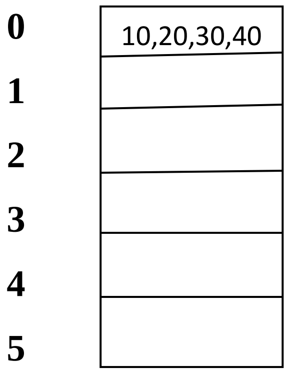
Example of collision in hashing –

Suppose we have elements 10,20,30,40 and we will insert these elements in a hash table using hash function h(x) %10. Where x is the element. So,

10%10 is equal to 0, and we place element 10 at index 0

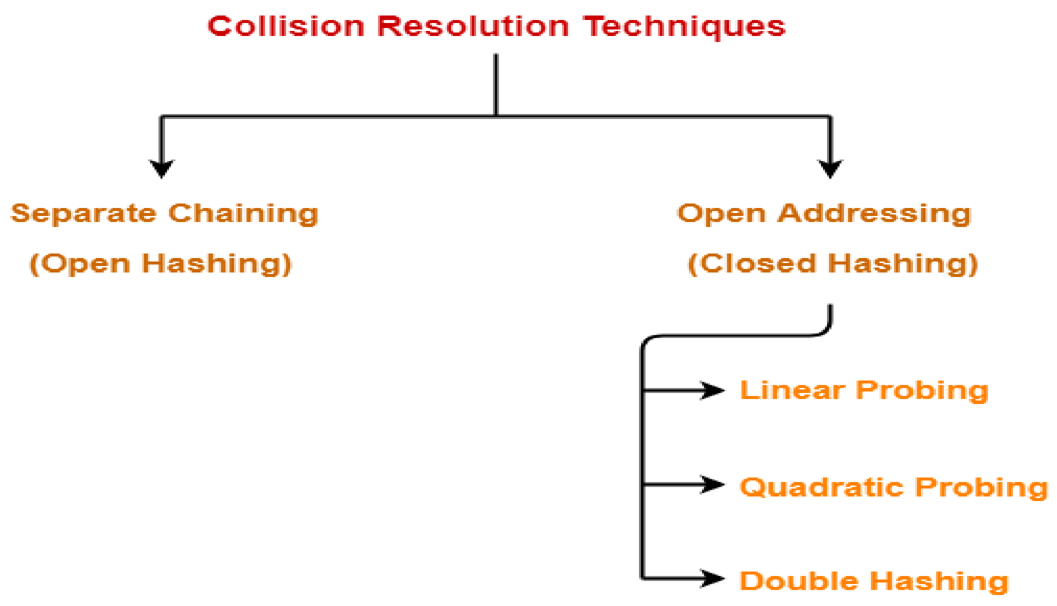
Similarly 20% 10 is 0, and we place element 10 at index 0

30%10 and 40 %10 all are 0, so we have to place all the elements at index 0. This is called collision in hashing.



<u>**Collision Resolution Techniques-**</u>

Collision resolution techniques are classified as-



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**.

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

1. Linear Probing-

In linear probing,

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

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.

Note : For linear search and binary search programs refer txt files.

Q. WAP in C to find an Element in an array using Linear Search.

```c
#include <stdio.h>

int main()
{
  int array[]= {20,40,70,90,10};
  int search,i;

  printf("Enter a number to search\n");
  scanf("%d", &search);

  for (i = 0; i < 5; i++)
  {
    if (array[i] == search)
    {
      printf("Number is present at location %d.\n", i);
      break;
    }
  }
  if (i == 5)
    printf("Number is not present in the array.\n");

  return 0;
}
```

Q. WAP in C to find an Element in an array using binary Search.

```c
#include <stdio.h>

int main()
{
   int first, last, middle, search;
   int array[]= {10,20,30,40,50};

   printf("Enter a number to search\n");
   scanf("%d", &search);

   first = 0;
   last = 4;
   middle = (first+last)/2;

   while (first <= last)
   {
      if (array[middle] < search)
         first = middle + 1;
      else if (array[middle] == search)
     {
         printf("number found at location %d.\n",middle);
         break;
      }
      else
         last = middle - 1;
      middle = (first + last)/2;
   }
   if (first > last)
      printf("Number is not present in the list.\n");

   return 0;
}
```