Program On Constructor in java

1) **Create a class named 'Rectangle' with two data members- length and breadth and a method to calculate the area which is 'length*breadth'. The class has three constructors which are :**
1 - having no parameter - values of both length and breadth are assigned zero.
2 - having two numbers as parameters - the two numbers are assigned as length and breadth respectively.
3 - having one number as parameter - both length and breadth are assigned that number.
Now, create objects of the 'Rectangle' class having none, one and two parameters and print their areas.

2) **Suppose you have a Piggie Bank with an initial amount of $50 and you have to add some more amount to it. Create a class 'AddAmount' with a data member named 'amount' with an initial value of $50. Now make two constructors of this class as follows:**
1 - without any parameter - no amount will be added to the Piggie Bank
2 - having a parameter which is the amount that will be added to Piggie Bank
Create object of the 'AddAmount' class and display the final amount in Piggie Bank.

3) **Create a class named 'Programming'. While creating an object of the class, if nothing is passed to it, then the message "I love programming languages" should be printed. If some String is passed to it, then in place of "programming languages" the name of that String variable should be printed.**
For example, while creating object if we pass "Java", then "I love Java" should be printed.

4) create the class name as Salary with a following constructor and methods

The major purpose of this program is calculate the salary of employee as per his present days using a constructor.

```
class Salary
{
  Salary(String name,int id,String contact,int presentDays,int perdaySalary)
   {  //here store the all data in instance variable of class
   }
  void calculateSalary()
   { //here need to write the logic for calculate the salary of employee as per his persent days
    }
  void show()
  {//here show the all details of employee
  }
}
class SalaryApp
{
   public static void main(String x[])
   { //here we need to create the object of Salary class and pass the values in its constructor
     // call the calculateSalary() function and show() function
   }
}
```

5) WAP to Encrypt the string using a constructor

Here we need to create the class name as Encrypt with a following constructor and method.

Input string : abcdabcdccddaabb

Ouput : a4b4c4d4

```
class Encrypt
{  Encrypt(String string)
    {
      //write here encryption logics
    }
  void beforeEncrypt()
   { //show here original string
   }
  void afterEncrypt()
  { //show here string after encrypt
  }
}
public class EncryptApp
{
   public static void main(String x[ ])
   {
      //here we need to create the object of Encrypt class and pass string in it
     //here call the function beforeEncrypt for show original string
     //here call the function afterEncrypt for show encrypted string.
   }
}
```

6.WAP for program for create the SingleTone class ?

Sorting theory

Introduction to Sorting

Sorting is nothing but arranging the data in ascending or descending order. The term **sorting** came into picture, as humans realised the importance of searching quickly.

There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc. All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of **sorting** came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search.

Sorting arranges data in a sequence which makes searching easier.

Sorting Efficiency

If you ask me, how will I arrange a deck of shuffled cards in order, I would say, I will start by checking every card, and making the deck as I move on.

It can take me hours to arrange the deck in order, but that's how I will do it.

Well, thank god, computers don't work like this.

Since the beginning of the programming age, computer scientists have been working on solving the problem of sorting by coming up with various different algorithms to sort data.

The two main criterias to judge which algorithm is better than the other have been:

1. Time taken to sort the given data.
2. Memory Space required to do so.

Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next few tutorials.

Bubble Sort

Insertion Sort

Selection Sort

Quick Sort

Merge Sort

Heap Sort

7.WAP to create the class name as BubbleSort with a following a constructor and methods

```
class BubbleSort
{   int a[]
     BubbleSort(int x[])
     {a=x; //here store the local array to instance variable
     }
    void beforeSort()
    { //show here original sorting
    }
   void performSorting()
    { apply here bubble sort logics
    }
   void afterSort()
    {//show here after sorting
    }
}
public class BubbleSortApp
{  public static void main(String x[])
    { //create the object of BubbleSort and pass array with a five values
      //call the before sort method for show original method
     // call the performSorting() and write the bubble sort logics
    //call the afterSort method for show the sorted array
    }
}
```

Refer Following Tutorial for bubble sort logics

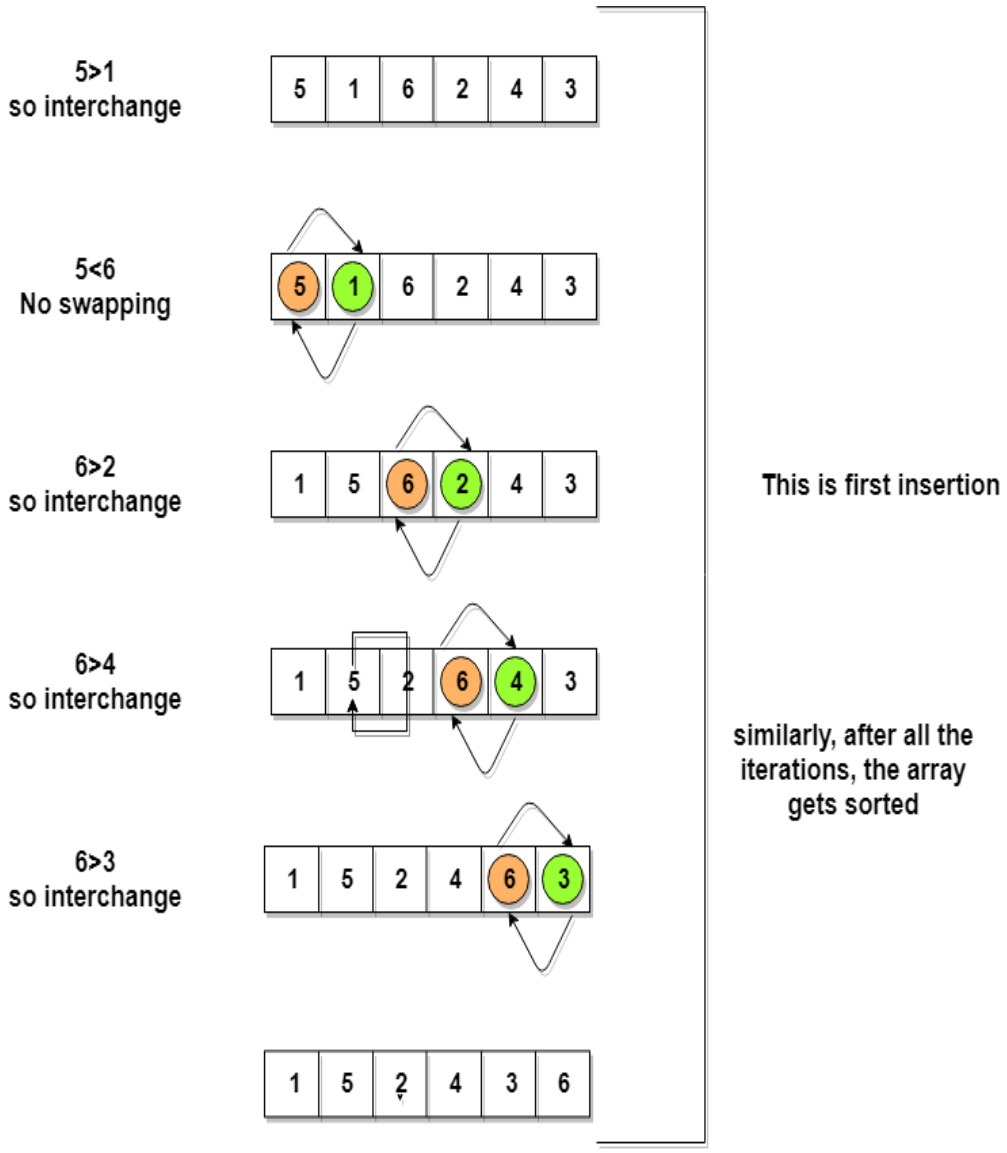
Implementing Bubble Sort Algorithm

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element(index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. **Repeat Step 1**.

Let's consider an array with values {5, 1, 6, 2, 4, 3}

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



So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.

Similarly after the second iteration, 5 will be at the second last index, and so on.

Although the above logic will sort an unsorted array, still the above algorithm is not efficient because as per the above logic, the outer for loop will keep on executing for **6** iterations even if the array gets sorted after the second iteration.

So, we can clearly optimize our algorithm.

Optimized Bubble Sort Algorithm(Second technique)

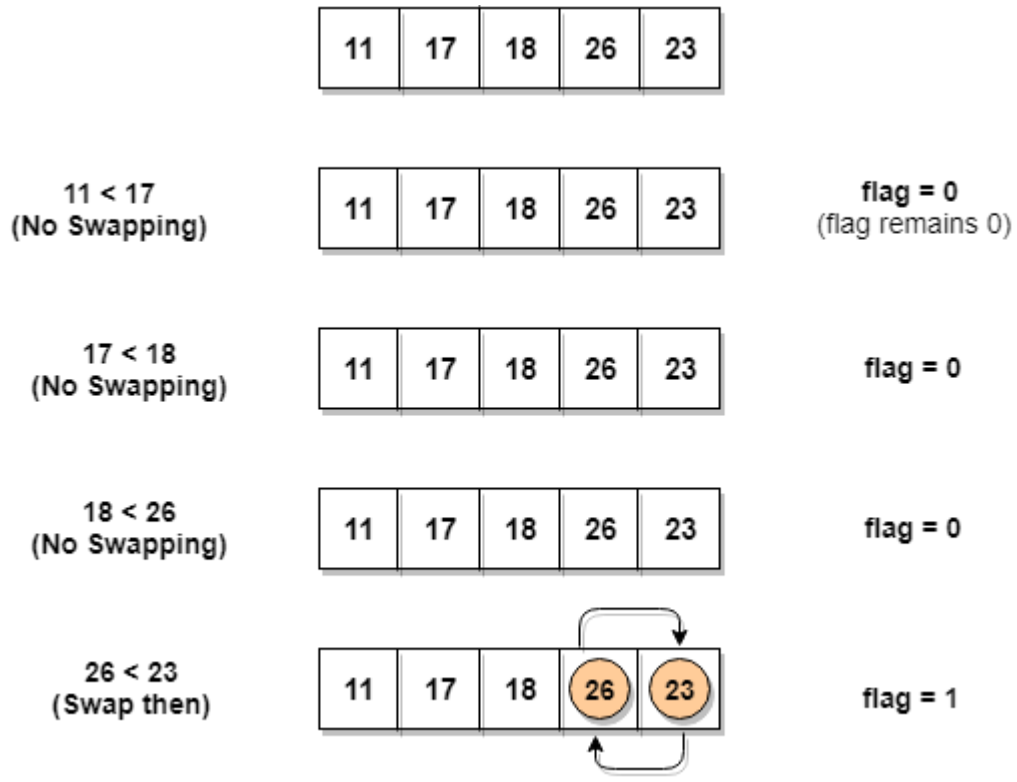
To optimize our bubble sort algorithm, we can introduce a flag to monitor whether elements are getting swapped inside the inner for loop.

Hence, in the inner for loop, we check whether swapping of elements is taking place or not, everytime.

If for a particular iteration, no swapping took place, it means the array has been sorted and we can jump out of the for loop, instead of executing all the iterations.

Let's consider an array with values {11, 17, 18, 26, 23}

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



In the above code, in the function bubbleSort, if for a single complete cycle of j iteration(inner for loop), no swapping takes place, then flag will remain 0 and then we will break out of the for loops, because the array has already been sorted.

Complexity Analysis of Bubble Sort

In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be,

(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1

Sum = n(n-1)/2

i.e $O(n^2)$

Hence the **time complexity** of Bubble Sort is **$O(n^2)$**.

The main advantage of Bubble Sort is the simplicity of the algorithm.

The **space complexity** for Bubble Sort is **O(1)**, because only a single additional memory space is required i.e. for temp variable.

Also, the **best case time complexity** will be **O(n)**, it is when the list is already sorted.

Following are the Time and Space complexity for the Bubble Sort algorithm.

- Worst Case Time Complexity [Big-O]: **$O(n^2)$**
- Best Case Time Complexity [Big-omega]: **O(n)**
- Average Time Complexity [Big-theta]: **$O(n^2)$**
- Space Complexity: **O(1)**

8.WAP to create the class name as SelectionSort with a following constructor and methods ?

```
class SelectionSort
{
    int a[ ];
    SelectionSort(int x[ ])
    {a=x;
    }
   void beforeSort()
   { //show here array before sorting
   }
   void performSorting()
   { //write here logics of selection sorting
   }
  void afterSort()
   { //show here logics after sorting
   }
}
public class SelectionSortApp
{  public static void main(String x[])
   {  //create here object of SelectionSort and pass array with a 5 size
     // call the beforeSort() method for a show the array without sorting
   //call here function performSorting() for performing sorting logics
   //call here function afterSort  for show the sorted array
  }
}
```

Refer following tutorial for Selection Sort

Selection Sort Algorithm

Selection sort is conceptually the most simplest sorting algorithm. This algorithm will first find the **smallest** element in the array and swap it with the element in the **first** position, then it will find the **second smallest** element and swap it with the element in the **second** position, and it will keep on doing this until the entire array is sorted.

It is called selection sort because it repeatedly **selects** the next-smallest element and swaps it into the right place.

How Selection Sort Works?

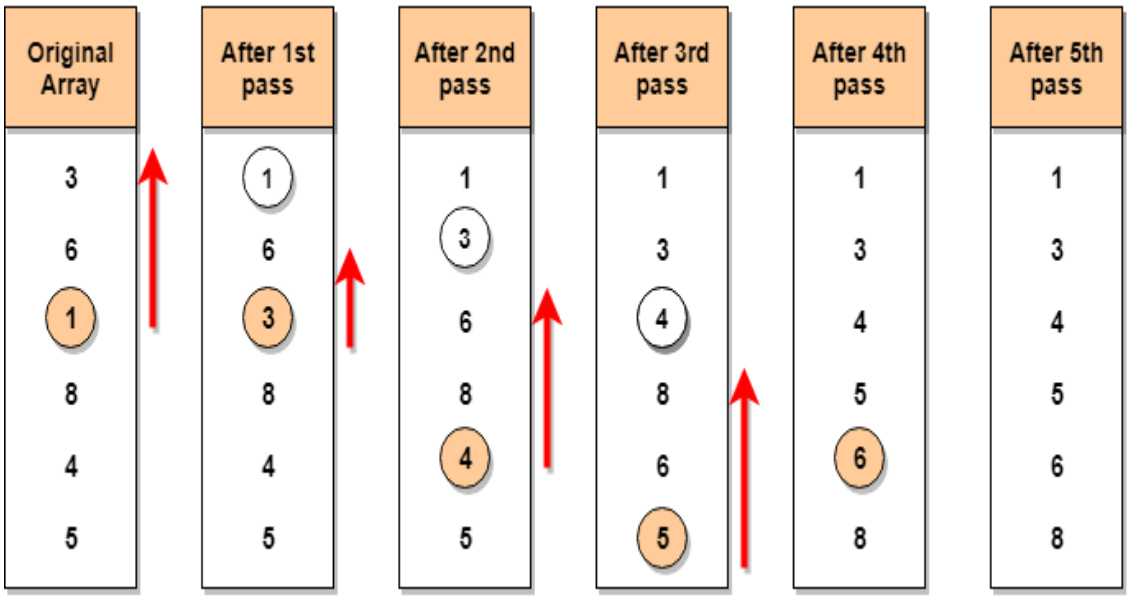
Following are the steps involved in selection sort(for sorting a given array in ascending order):

1. Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
2. We then move on to the second position, and look for smallest element present in the subarray, starting from index 1, till the last index.
3. We replace the element at the **second** position in the original array, or we can say at the first position in the subarray, with the second smallest element.

4. This is repeated, until the array is completely sorted.

Let's consider an array with values {3, 6, 1, 8, 4, 5}

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



In the **first** pass, the smallest element will be 1, so it will be placed at the first position.

Then leaving the first element, **next smallest** element will be searched, from the remaining elements. We will get 3 as the smallest, so it will be then placed at the second position.

Then leaving 1 and 3(because they are at the correct position), we will search for the next smallest element from the rest of the elements and put it at third position and keep doing this until array is sorted.

Finding Smallest Element in a subarray

In selection sort, in the first step, we look for the smallest element in the array and replace it with the element at the first position. This seems doable, isn't it?

Consider that you have an array with following values {3, 6, 1, 8, 4, 5}. Now as per selection sort, we will start from the first element and look for the smallest number in the array, which is 1 and we will find it at the **index** 2. Once the smallest number is found, it is swapped with the element at the first position.

Well, in the next iteration, we will have to look for the second smallest number in the array. How can we find the second smallest number? This one is tricky?

If you look closely, we already have the smallest number/element at the first position, which is the right position for it and we do not have to move it anywhere now. So we can say, that the first element is sorted, but the elements to the right, starting from index 1 are not.

So, we will now look for the smallest element in the subarray, starting from index 1, to the last index.

Confused? Give it time to sink in.

After we have found the second smallest element and replaced it with element on index 1(which is the second position in the array), we will have the first two positions of the array sorted.

Then we will work on the subarray, starting from index 2 now, and again looking for the smallest element in this subarray.

Complexity Analysis of Selection Sort

Selection Sort requires two nested for loops to complete itself, one for loop is in the function selectionSort, and inside the first loop we are making a call to another function indexOfMinimum, which has the second(inner) for loop.

Hence for a given input size of n, following will be the time and space complexity for selection sort algorithm:

Worst Case Time Complexity [Big-O]: **$O(n^2)$**

Best Case Time Complexity [Big-omega]: **$O(n^2)$**

Average Time Complexity [Big-theta]: **$O(n^2)$**

Space Complexity: **O(1)**

9.WAP to create the class name as InsertionSort with a following constructor and methods

```
class InsertionSort
{   int a[ ];
     InsertionSort(int x[ ])
     {a=x;//store here local array to instance variable
     }
    void beforeSort()
    {//write here logics without sorting array
    }
   void performSorting()
    { //write here logics for insertion sorting
    }
   void afterSort()
    {//show array here after sorting
    }
}
public class InsertionSortApp
{  public static void main(String x[ ])
    {  //create the object of InsertionSort and pass array in it
      // call the beforeSort() method and show the original array
      //call the performSorting() method and for execute the insertion sort logics
     //call the afterSort() method for display the sorted array
    }
}
```

Refer Following tutorial for Insertion Sorting Purpose

Consider you have 10 cards out of a deck of cards in your hand. And they are sorted, or arranged in the ascending order of their numbers.

If I give you another card, and ask you to **insert** the card in just the right position, so that the cards in your hand are still sorted. What will you do?

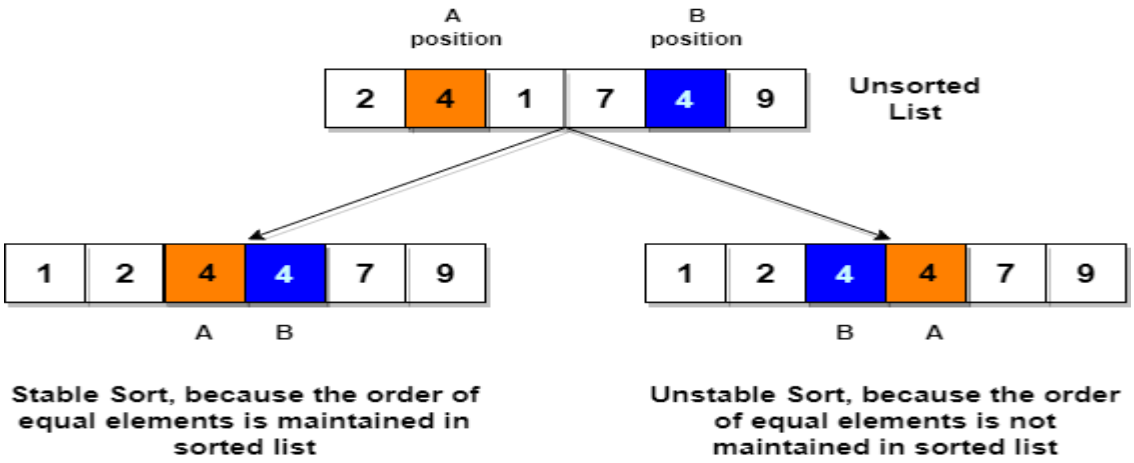
Well, you will have to go through each card from the starting or the back and find the right position for the new card, comparing it's value with each card. Once you find the right position, you will **insert** the card there.

Similarly, if more new cards are provided to you, you can easily repeat the same process and insert the new cards and keep the cards sorted too.

This is exactly how **insertion sort** works. It starts from the index 1(not 0), and each index starting from index 1 is like a new card, that you have to place at the right position in the sorted subarray on the left.

Following are some of the important **characteristics of Insertion Sort**:

1. It is efficient for smaller data sets, but very inefficient for larger lists.
2. Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
3. It is better than Selection Sort and Bubble Sort algorithms.
4. Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
5. It is a **stable** sorting technique, as it does not change the relative order of elements which are equal.



How Insertion Sort Works?

Following are the steps involved in insertion sort:

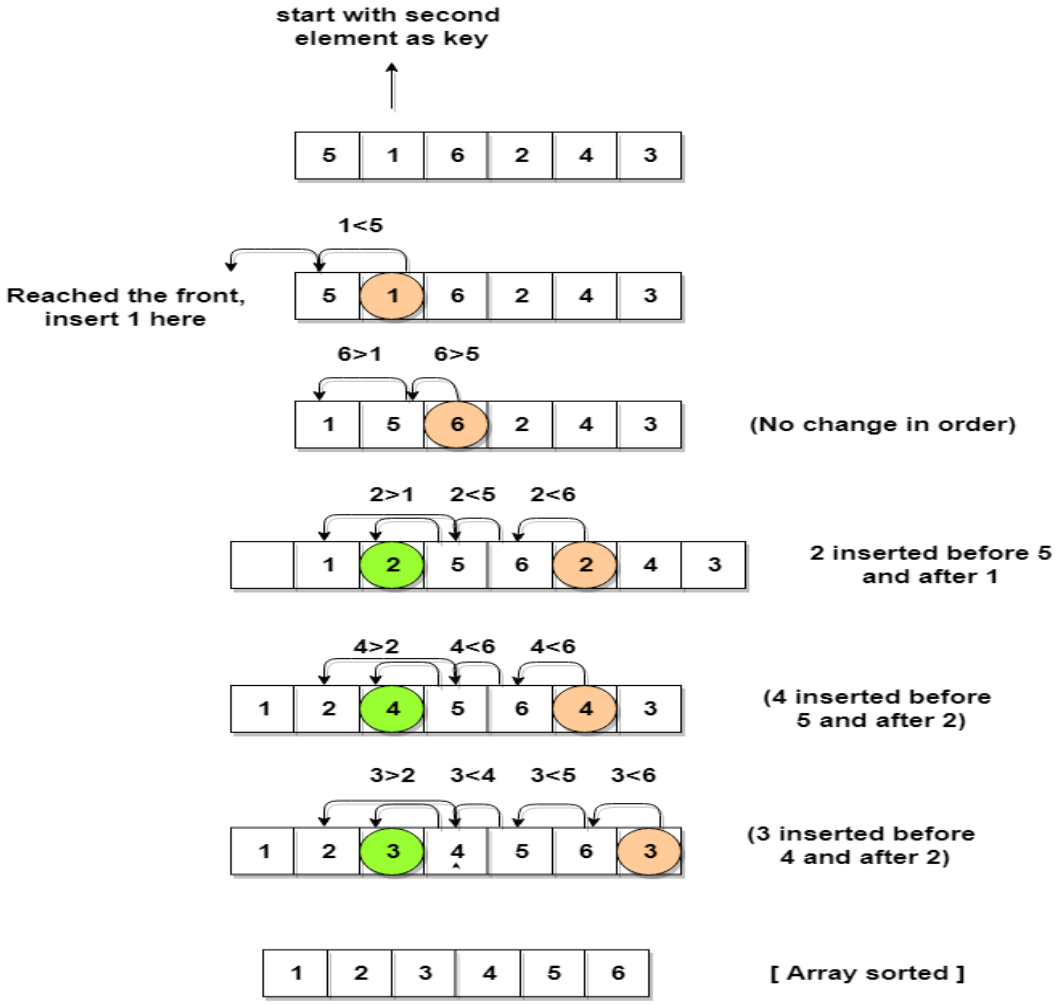
1. We start by making the second element of the given array, i.e. element at index 1, the key. The key element here is the new card that we need to add to our existing sorted set of cards(remember the example with cards above).

2. We compare the key element with the element(s) before it, in this case, element at index 0:
    - If the key element is less than the first element, we insert the key element before the first element.
    - If the key element is greater than the first element, then we insert it after the first element.
3. Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the right position.
4. And we go on repeating this, until the array is sorted.

Let's consider an array with values {5, 1, 6, 2, 4, 3}

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

Refer diagram.



As you can see in the diagram above, after picking a key, we start iterating over the elements to the left of the key.

We continue to move towards left if the elements are greater than the key element and stop when we find the element which is less than the key element.

Complexity Analysis of Insertion Sort

As we mentioned above that insertion sort is an efficient sorting algorithm, as it does not run on preset conditions using for loops, but instead it uses one while loop, which avoids extra steps once the array gets sorted.

Even though insertion sort is efficient, still, if we provide an already sorted array to the insertion sort algorithm, it will still execute the outer for loop, thereby requiring n steps to sort an already sorted array of n elements, which makes its **best case time complexity** a linear function of n.

Worst Case Time Complexity [Big-O]: **$O(n^2)$**

Best Case Time Complexity [Big-omega]: **O(n)**

Average Time Complexity [Big-theta]: **$O(n^2)$**. Space Complexity: **O(1)**

10) WAP to create the class name as MergeSort with a following constructor and methods

```
class MergeSort
{    int a[];
     MergeSort(int x[ ])
     {a=x;//store here array of local variable in to the instance variable
     }
     void beforeSort()
     {//show here original array
     }
    void performSorting()
    {//write here logics of merge sort and perform sorting operation on it
    }
   void afterSort()
   { //show here array after sorting
   }
}
public class MergeSortApp
{  public static void main(String x[])
   {  //create here object of MergeSort and pass array of size 5 element
     // call the beforeSort() method and show the original array
    //call the performSorting() method and write the sorting logics
   //call the afterSort() method and perform the array sorting example
   }
}
```

Refer Following tutorial for perform the merge sort application

Merge Sort Algorithm

Merge Sort follows the rule of **Divide and Conquer** to sort a given set of numbers/elements, recursively, hence consuming less time.

In the last two tutorials, we learned about Selection Sort and Insertion Sort, both of which have a worst-case running time of $O(n^2)$. As the size of input grows, insertion and selection sort can take a long time to run.

Merge sort , on the other hand, runs in O(n*log n) time in all the cases.

Before jumping on to, how merge sort works and it's implementation, first lets understand what is the rule of **Divide and Conquer**?

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**.

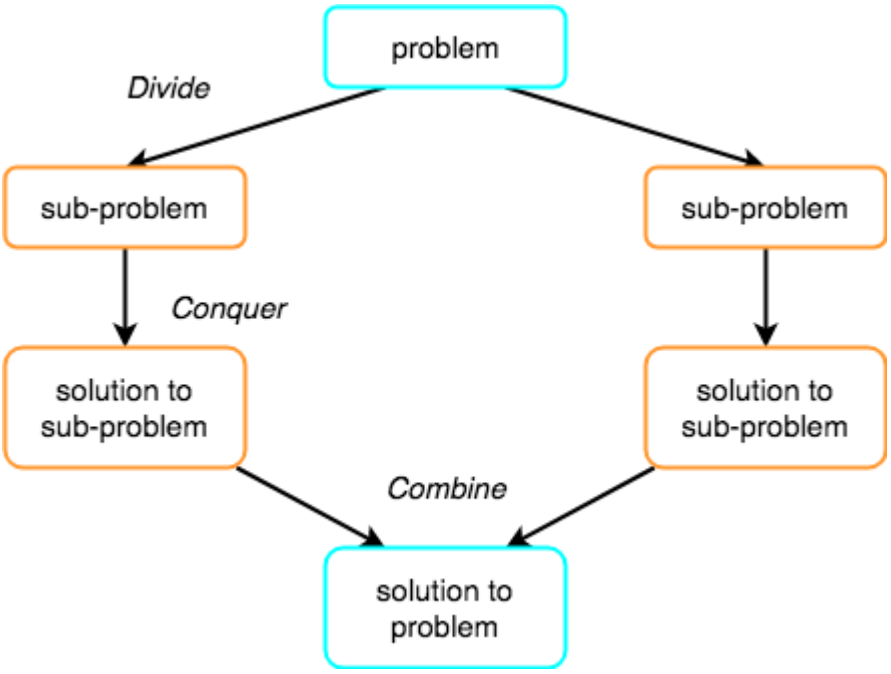
When Britishers came to India, they saw a country with different religions living in harmony, hard working but naive citizens, unity in diversity, and found it difficult to establish their empire. So, they adopted the policy of **Divide and Rule**. Where the population of India was collectively a one 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.

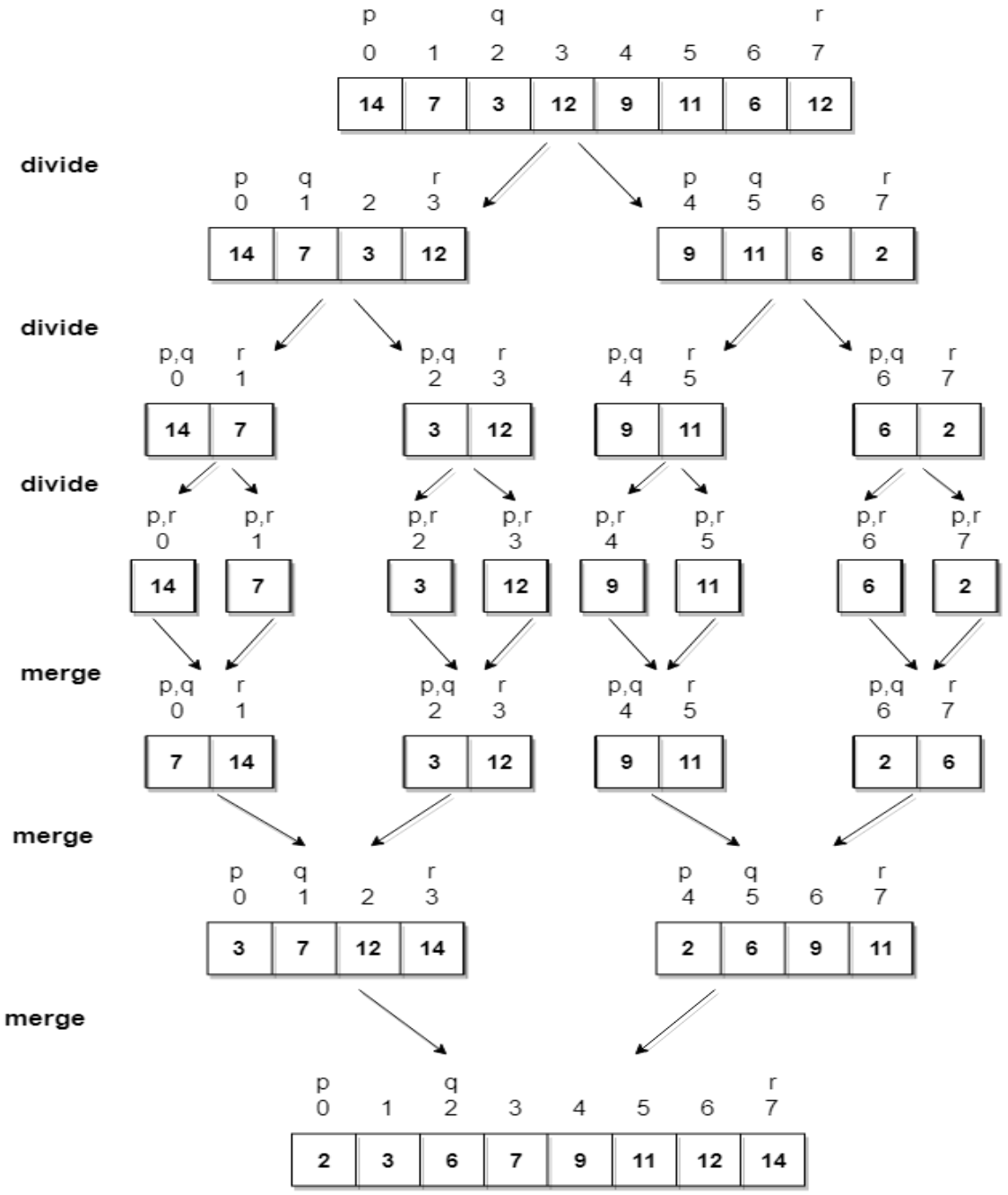
But breaking the orignal array into 2 smaller subarrays is not helping us in sorting the array.

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.

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.



n 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.

Complexity Analysis of Merge Sort

Merge Sort is quite fast, and has a time complexity of O(n*log n). It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

In this section we will understand why the running time for merge sort is O(n*log n).

As we have already learned in Binary Search that whenever we divide a number into half in every stpe, it can be represented using a logarithmic function, which is log n and the number of steps can be represented by log n + 1(at most)

Also, we perform a single step operation to find out the middle of any subarray, i.e. O(1).

And to **merge** the subarrays, made by dividing the original array of n elements, a running time of O(n) will be required.

Hence the total time for mergeSort function will become n(log n + 1), which gives us a time complexity of O(n*log n).

Worst Case Time Complexity [Big-O]: **O(n*log n)**

Best Case Time Complexity [Big-omega]: **O(n*log n)**

Average Time Complexity [Big-theta]: **O(n*log n)**

Space Complexity: **O(n)**

- Time complexity of Merge Sort is O(n*Log n) in all the 3 cases (worst, average and best) as merge sort always **divides** the array in two halves and takes linear time to **merge** two halves.
- It requires **equal amount of additional space** as the unsorted array. Hence its not at all recommended for searching large unsorted arrays.
- It is the best Sorting technique used for sorting **Linked Lists**.

11) WAP to create the class name as QuickSort with a following constructor and methods

```
class QuickSort
{    int a[ ];
      QuickSort(int x[ ])
      {
          a=x;
      }
      void beforeSort()
      {//write here original array
      }
     void performSort()
     {//here write logics to perform the quick sort
     }
    void afterSort()
     {//here show the sorted array  after sorting
    }
}
public class QuickSortApp
{  public static void main(String x[ ])
    {  //create the object of QuickSort class and pass array size 5 element
       // call the beforeSort() method and show the original array
      //call the performSort() method and write here quick sort logics
     //call the afterSort() method and show the sorted array
     }
}
```

Refer the Following Tutorial for QuickSortApp

Quick Sort Algorithm

Quick Sort is also based on the concept of **Divide and Conquer**, just like merge sort. But in quick sort all the heavy lifting(major work) is done while **dividing** the array into subarrays, while in case of merge sort, all the real work happens during **merging** the subarrays. In case of quick sort, the combine step does absolutely nothing.

It is also called **partition-exchange sort**. This algorithm divides the list into three main parts:

1. Elements less than the **Pivot** element
2. Pivot element(Central element)
3. Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as **pivot**.

For example: In the array {52, 37, 63, 14, 17, 8, 6, 25}, we take 25 as **pivot**. So after the first pass, the list will be changed like this.

{6 8 17 14 **25** 63 37 52}

Hence after the first pass, pivot will be set at its position, with all the elements **smaller** to it on its left and all the elements **larger** than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate subarrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted.

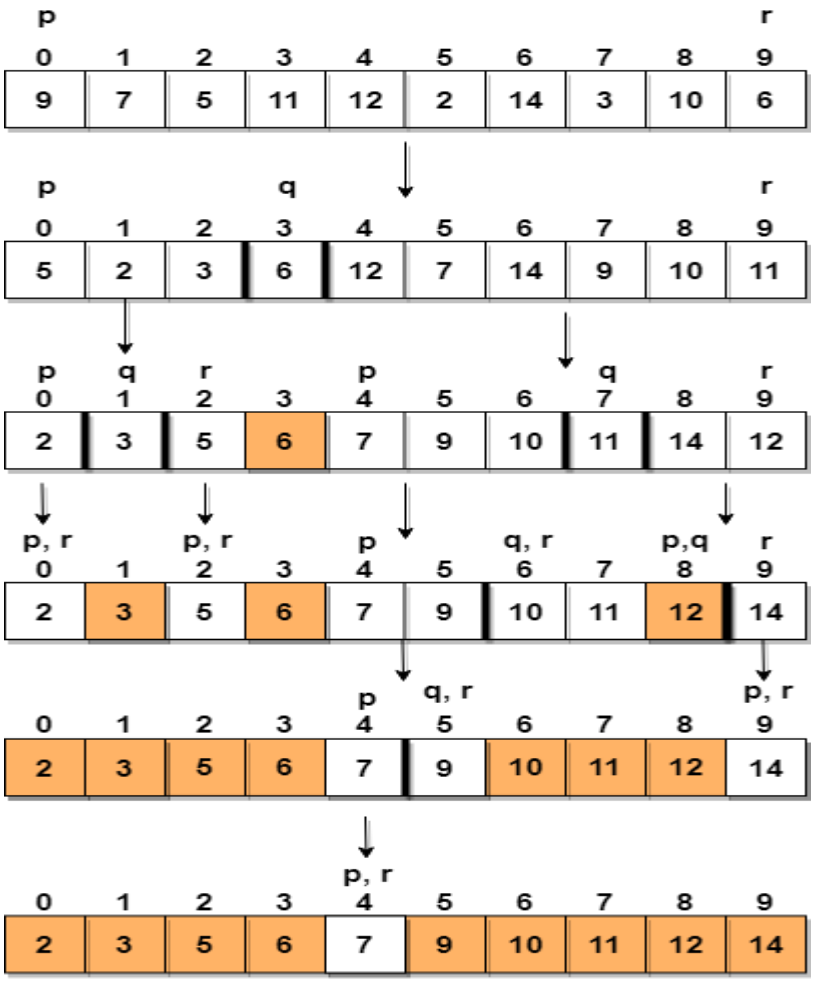
How Quick Sorting Works?

Following are the steps involved in quick sort algorithm:

1. After selecting an element as **pivot**, which is the last index of the array in our case, we divide the array for the first time.
2. In quick sort, we call this **partitioning**. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the **pivot** will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.
3. And the **pivot** element will be at its final **sorted** position.
4. The elements to the left and right, may not be sorted.
5. Then we pick subarrays, elements on the left of **pivot** and elements on the right of **pivot**, and we perform **partitioning** on them by choosing a **pivot** in the subarrays.

Let's consider an array with values {9, 7, 5, 11, 12, 2, 14, 3, 10, 6}

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



In step 1, we select the last element as the **pivot**, which is 6 in this case, and call for partitioning, hence re-arranging the array in such a way that 6 will be placed in its final position and to its left will be all the elements less than it and to its right, we will have all the elements greater than it.

Then we pick the subarray on the left and the subarray on the right and select a **pivot** for them, in the above diagram, we chose 3 as pivot for the left subarray and 11 as pivot for the right subarray.

And we again call for partitioning.

Complexity Analysis of Quick Sort

For an array, in which **partitioning** leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the **pivot**, hence on the right side.

And if keep on getting unbalanced subarrays, then the running time is the worst case, which is $O(n^2)$

Where as if **partitioning** leads to almost equal subarrays, then the running time is the best, with time complexity as **O(n*log n)**.

Worst Case Time Complexity [Big-O]: **O(n^2)**

Best Case Time Complexity [Big-omega]: **O(n*log n)**

Average Time Complexity [Big-theta]: **O(n*log n)**

Space Complexity: **O(n*log n)**

As we know now, that if subarrays **partitioning** produced after partitioning are unbalanced, quick sort will take more time to finish. If someone knows that you pick the last index as **pivot** all the time, they can intentionally provide you with array which will result in worst-case running time for quick sort.

To avoid this, you can pick random **pivot** element too. It won't make any difference in the algorithm, as all you need to do is, pick a random element from the array, swap it with element at the last index, make it the **pivot** and carry on with quick sort.

- Space required by quick sort is very less, only O(n*log n) additional space is required.
- Quick sort is not a stable sorting technique, so it might change the occurence of two similar elements in the list while sorting.

12) WAP to create the class with a following constructor and methods

```
class HeapSort
{  int a[];
     HeapSort(int x[])
     {a=x;//store here local array in to the instance array
     }
   void beforeSort()
   {//show here unsorted array
   }
  void performSorting()
   { //perform here quick sort algoritham
   }
  void afterSort()
   { //show here sorted array
   }
}
public class HeapSortApp
{  public static void main(String x[])
    {   //create the object of HeapSort and pass array in its constructor
       // call the beforeSort() method and for show the unsorted array
     //call the performSorting() method  for execute the sorting logics
     //call the afterSort() method for show the array after sorting
    }
}
```

Refer the following tutorial for Quick Sort Application

Heap Sort Algorithm

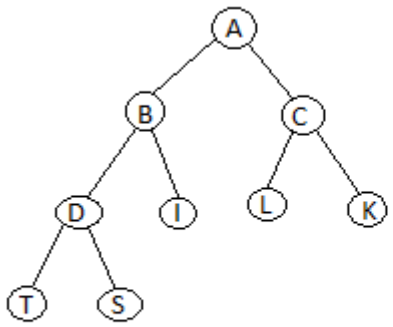
Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case running time. Heap sort involves building a **Heap** data structure from the given array and then utilizing the Heap to sort the array.

You must be wondering, how converting an array of numbers into a heap data structure will help in sorting the array. To understand this, let's start by understanding what is a Heap.

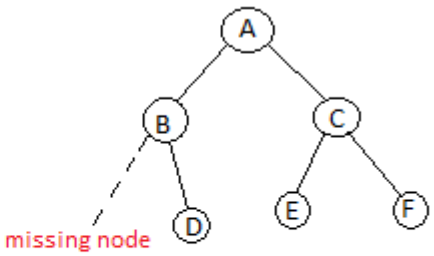
What is a Heap ?

Heap is a special tree-based data structure, that satisfies the following special heap properties:

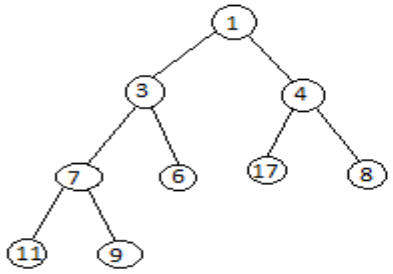
1. **Shape Property:** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.



Complete Binary Tree

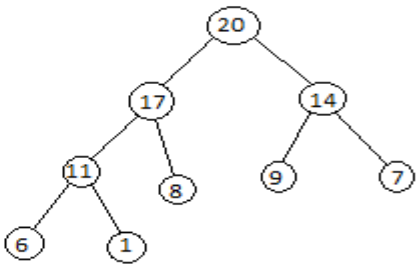


In-Complete Binary Tree



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

How Heap Sort Works?

Heap sort algorithm is divided into two basic parts:

- Creating a Heap of the unsorted list/array.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure(Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest(depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.

Complexity Analysis of Heap Sort

Worst Case Time Complexity: **O(n*log n)**

Best Case Time Complexity: **O(n*log n)**

Average Time Complexity: **O(n*log n)**

Space Complexity : **O(1)**

- Heap sort is not a Stable sort, and requires a constant space for sorting a list.
- Heap Sort is very fast and is widely used for sorting.