DATA STRUCTURES

LECTURE-17

HEAP

Dr. Sumitra Kisan

A **heap** is a complete binary tree-based **data structure** that follows the **heap property**. In a heap, the value of each node is compared to the values of its children in a specific way:

- **Max-Heap**: The value of each node is greater than or equal to the values of its children, ensuring that the root node contains the maximum value. As you move down the tree, the values decrease. The root node contains the maximum value, and the values decrease as you move down the tree.

  A[Parent(i)] >= A[i]

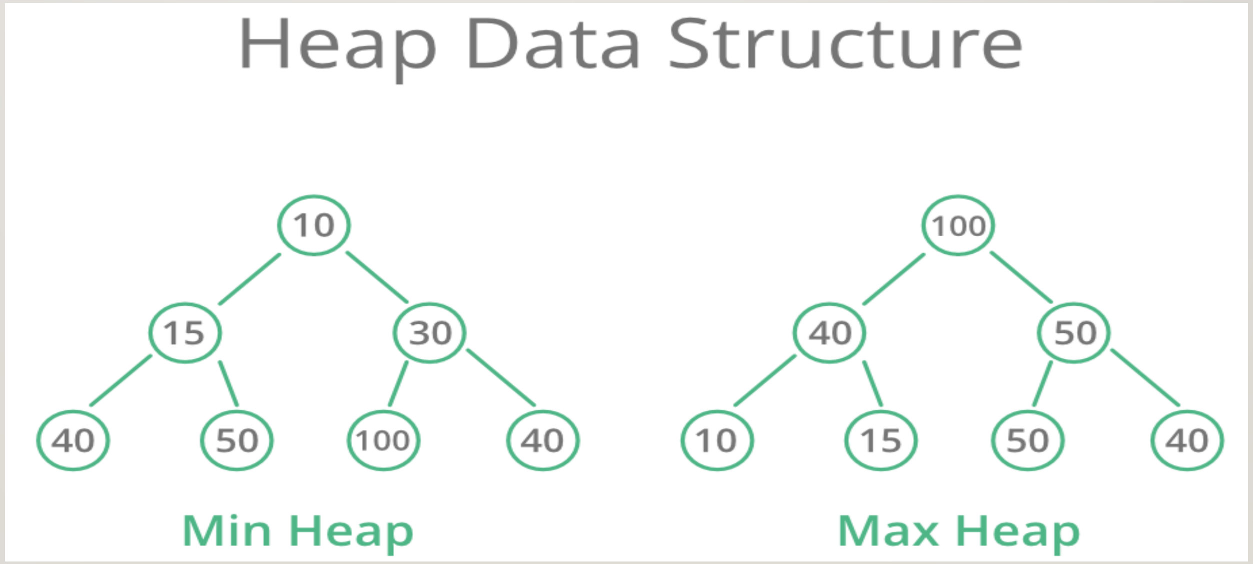
- **Min-Heap**: The value of each node is less than or equal to the values of its children, ensuring that the root node contains the minimum value. As you move down the tree, the values increase. The root node contains the minimum value, and the values increase as you move down the tree.

  A[Parent(i)] <= A[i]

Heap Operations

Common heap operations are:

- **Insert**: Adds a new element to the heap while maintaining the heap property.
- **Extract Max/Min:** Removes the maximum or minimum element from the heap and returns it.
- **Heapify**: Converts an arbitrary binary tree into a heap.



Time complexity in Max Heap

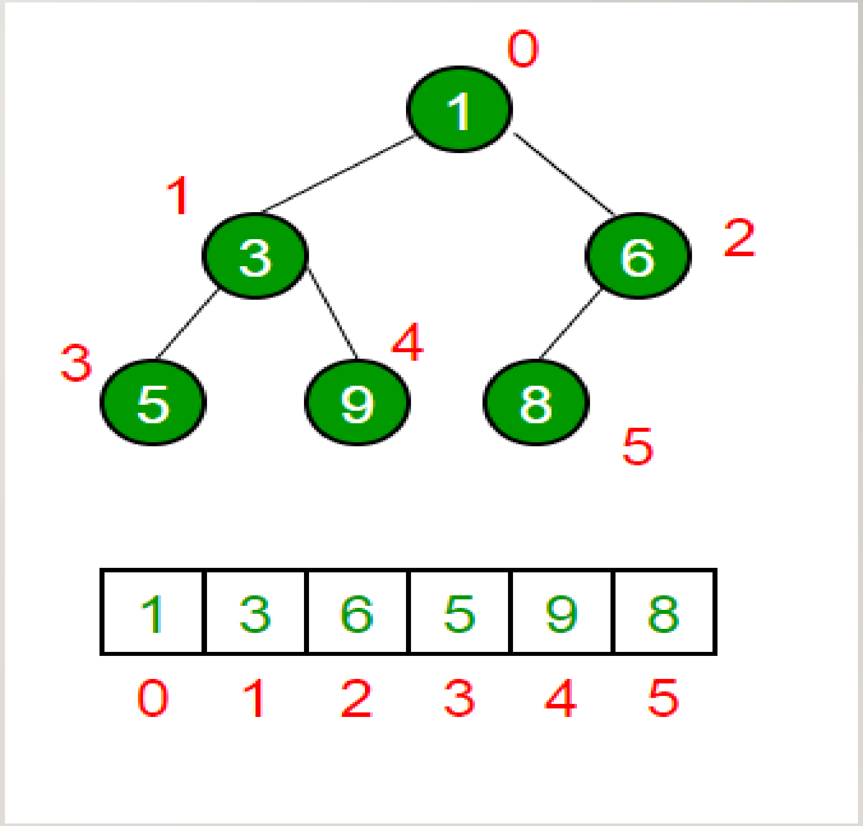
The total number of comparisons required in the max heap is according to the height of the tree. The height of the complete binary tree is always logn; therefore, the time complexity would also be O(logn).

Array Representation Of Binary Heap

The representation is done as:

- The root element will be at Arr[0].
- Below table shows indexes of other nodes for the i^{th} node, i.e., Arr[i]

Arr[(i-1)/2]	Returns the parent node
Arr[(2*i)+1]	Returns the left child node
Arr[(2*i)+2]	Returns the right child node



Algorithm of insert operation in the max heap

```
/ algorithm to insert an element in the max heap.
insertHeap(A, n, value)
{
n=n+1; // n is incremented to insert the new element
A[n]=value; // assign new value at the nth position
i = n; // assign the value of n to i
// loop will be executed until i becomes 1.
while(i>1)
{   parent= floor value of i/2; // Calculating the floor value of i/2
// Condition to check whether the value of parent is less than the given node or not
if(A[parent]<A[i])
{
swap(A[parent], A[i]);
i = parent;  }
else
{
return;
}  }  }
```

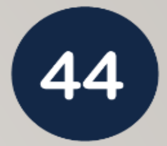
Example:

Insertion in the Heap tree: **44, 33, 77, 11, 55, 88, 66**

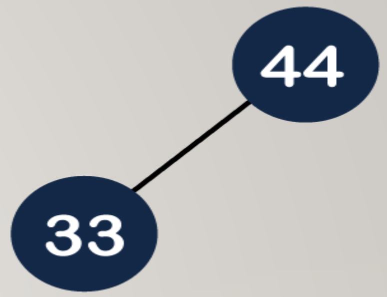
Suppose we want to create the max heap tree. To create the max heap tree, we need to consider the following two cases:

- First, we have to insert the element in such a way that the property of the complete binary tree must be maintained.
- Secondly, the value of the parent node should be greater than the either of its child.

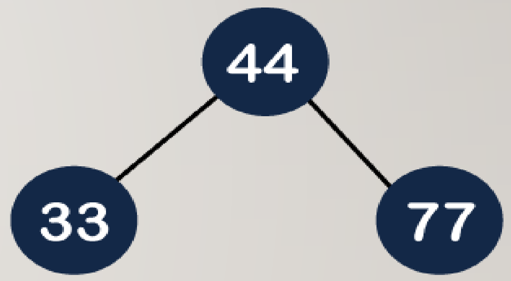
Step 1: First we add the 44 element in the tree as shown below:



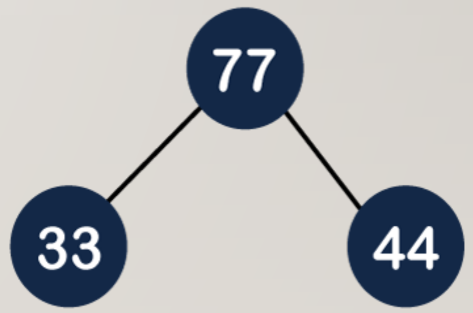
Step 2: The next element is 33. As we know that insertion in the binary tree always starts from the left side so 33 will be added at the left of 44 as shown below:



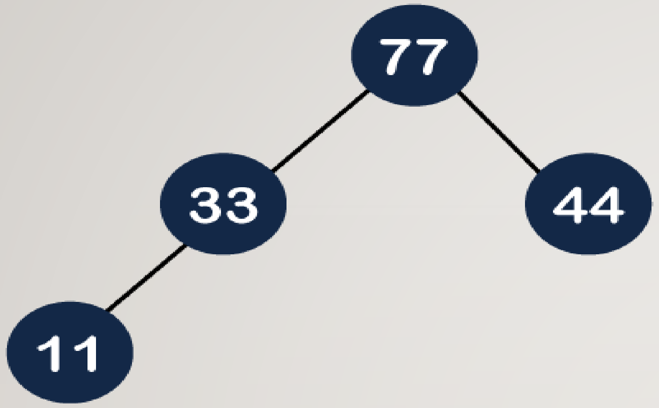
Step 3: The next element is 77 and it will be added to the right of the 44 as shown below:



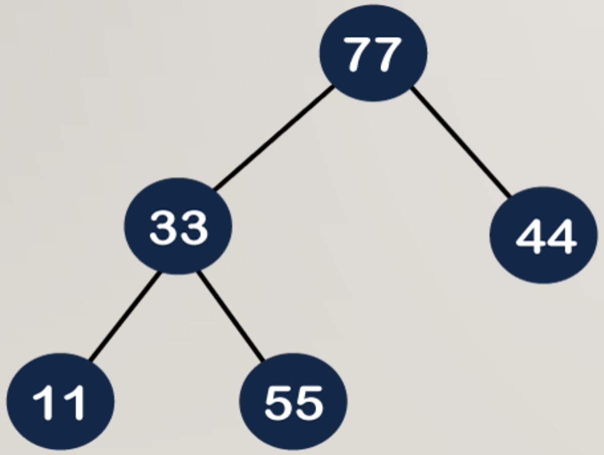
As we can observe in the above tree that it does not satisfy the max heap property, i.e., parent node 44 is less than the child 77. So, we will swap these two values as shown below:



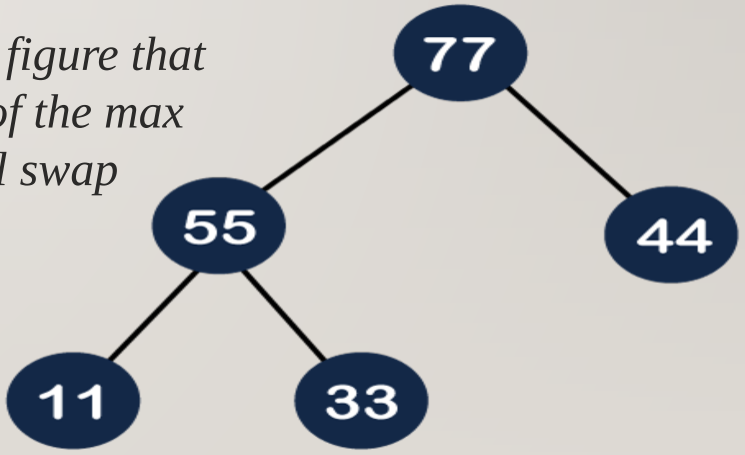
Step 4: The next element is 11. The node 11 is added to the left of 33 as shown below:



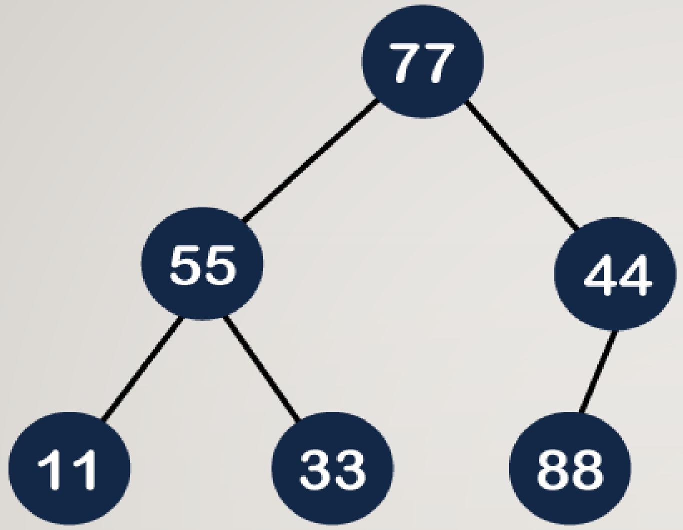
Step 5: The next element is 55. To make it a complete binary tree, we will add the node 55 to the right of 33 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because 33<55, so we will swap these two values as shown:



Step 6: The next element is 88. The left subtree is completed so we will add 88 to the left of 44 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because 44<88, so we will swap these two values as shown below:
Again, it is violating the max heap property because 88>77 so we will swap these two values as shown below:

Step 7: The next element is 66. To make a complete binary tree, we will add the 66 element to the right side of 77 as shown below:

Working of Heap sort Algorithm

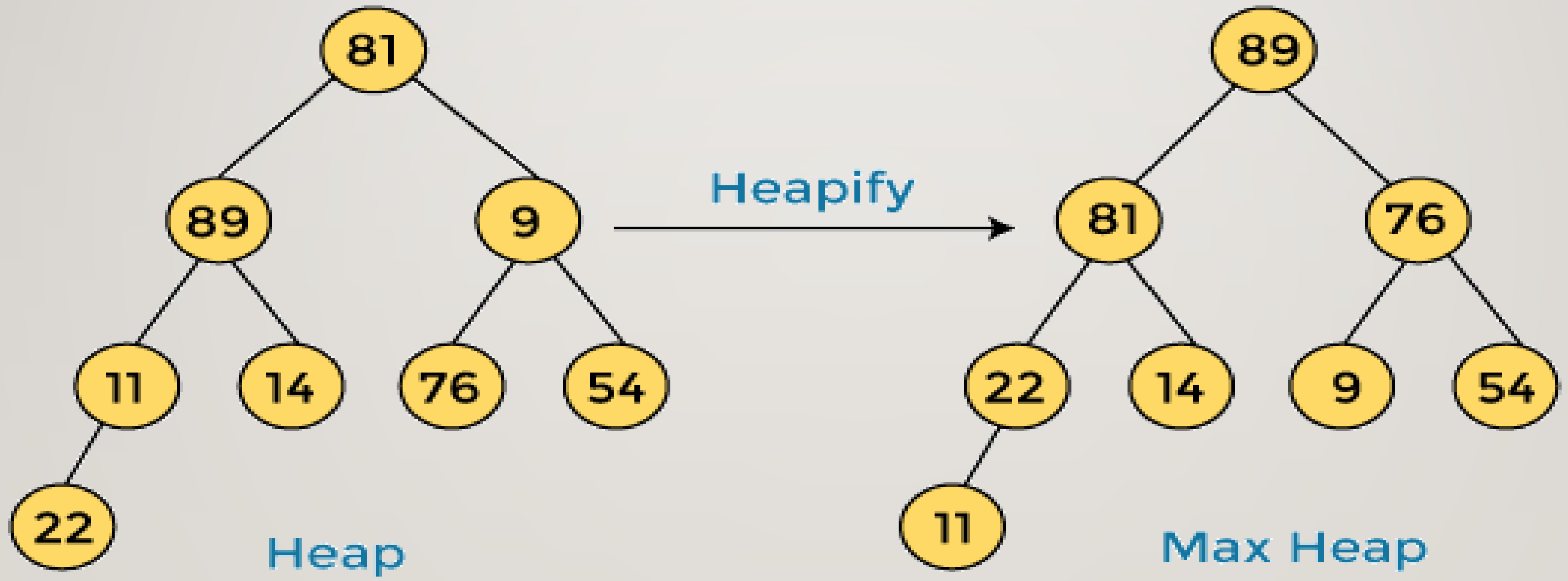
In heap sort, basically, there are two phases involved in the sorting of elements.

- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

Consider the following list of elements:

81	89	9	11	14	76	54	22

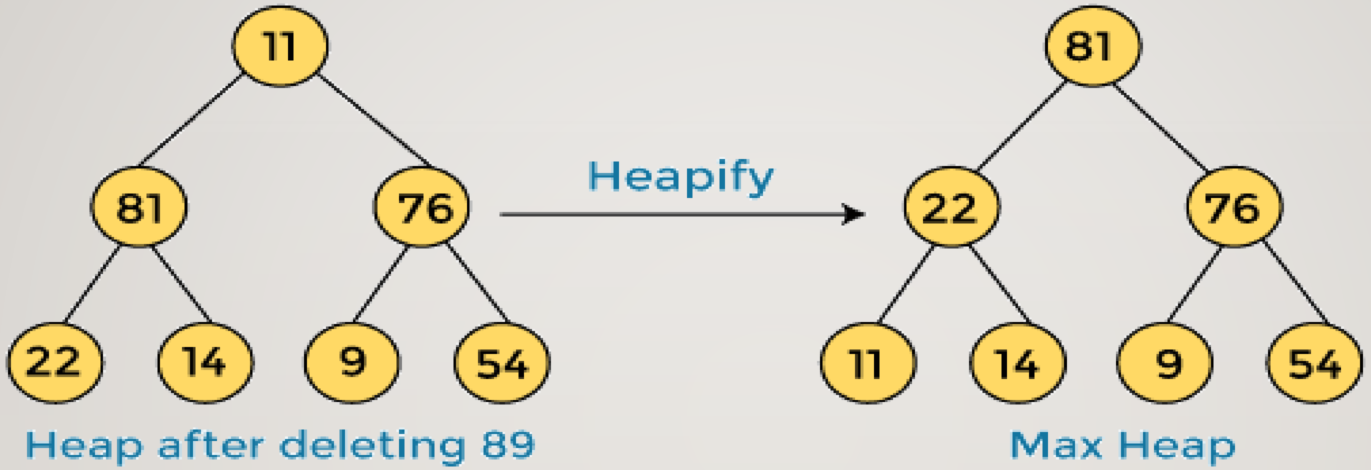
First, we have to construct a heap from the given array and convert it into max heap.



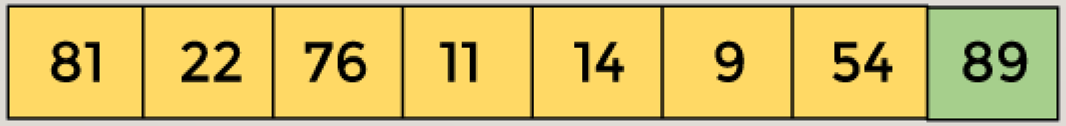
After converting the given heap into max heap, the array elements are -

89	81	76	22	14	9	54	11

Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.

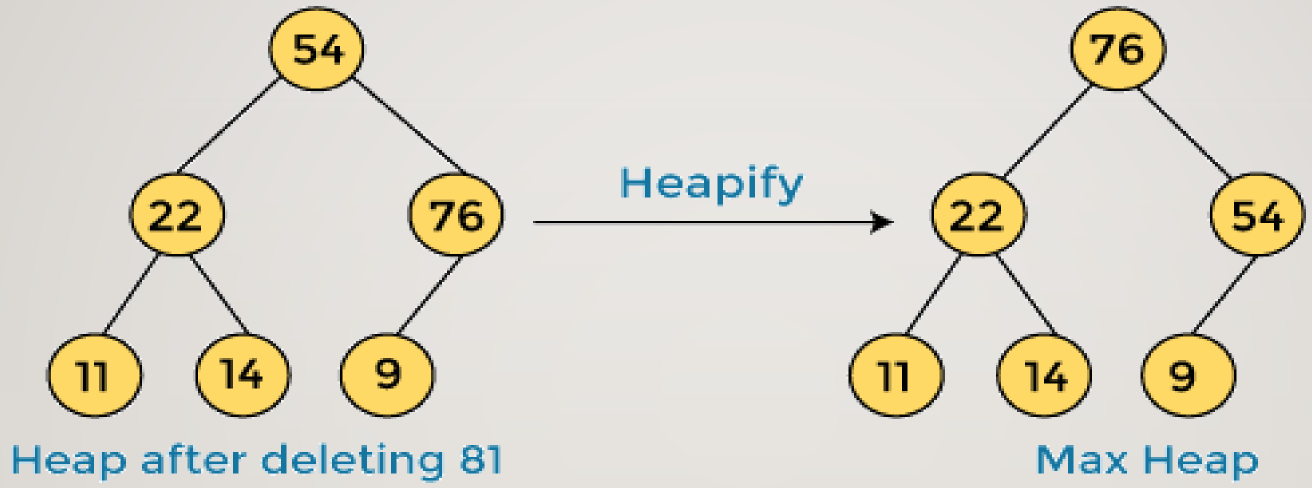


After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -

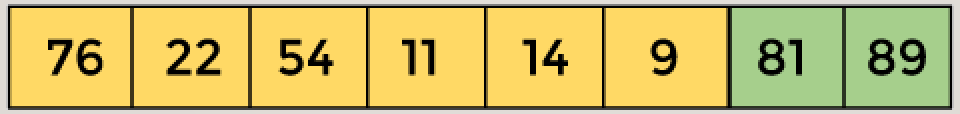


81	22	76	11	14	9	54	89

In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.

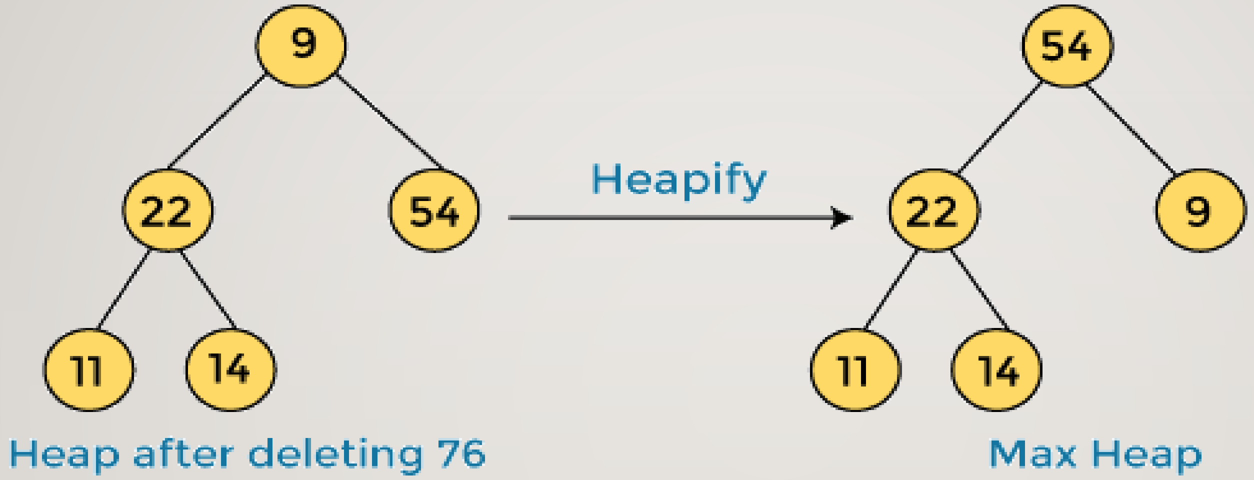


After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -

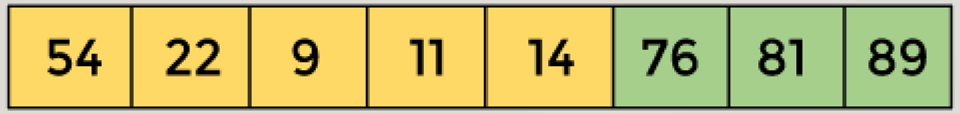


76	22	54	11	14	9	81	89

In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

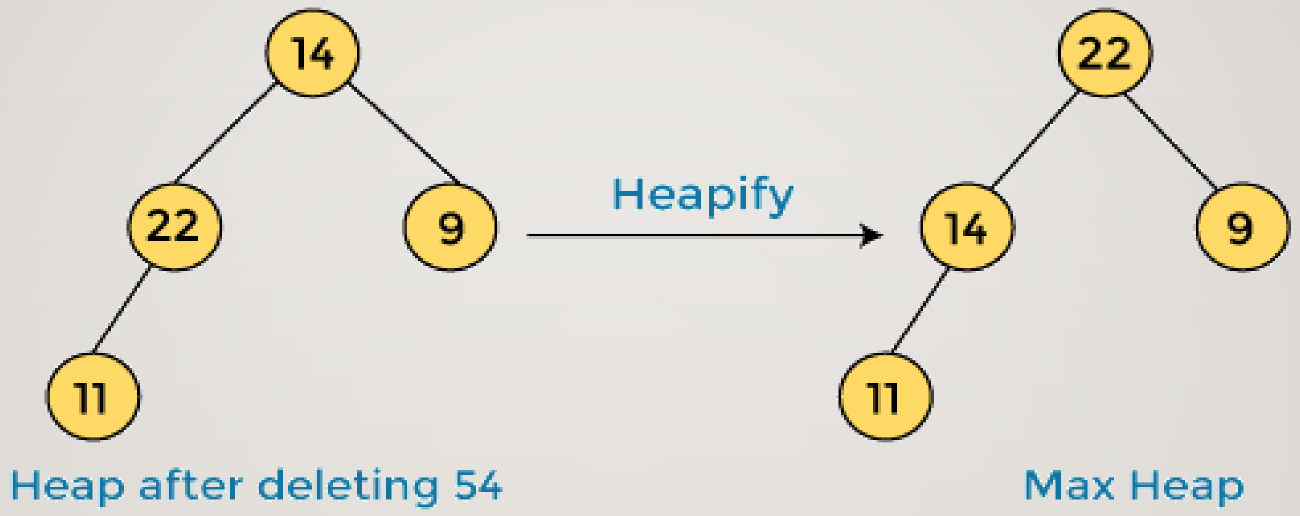


After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -

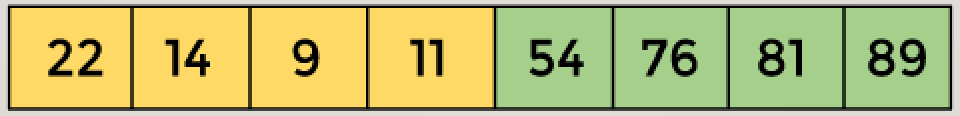


54	22	9	11	14	76	81	89

In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.

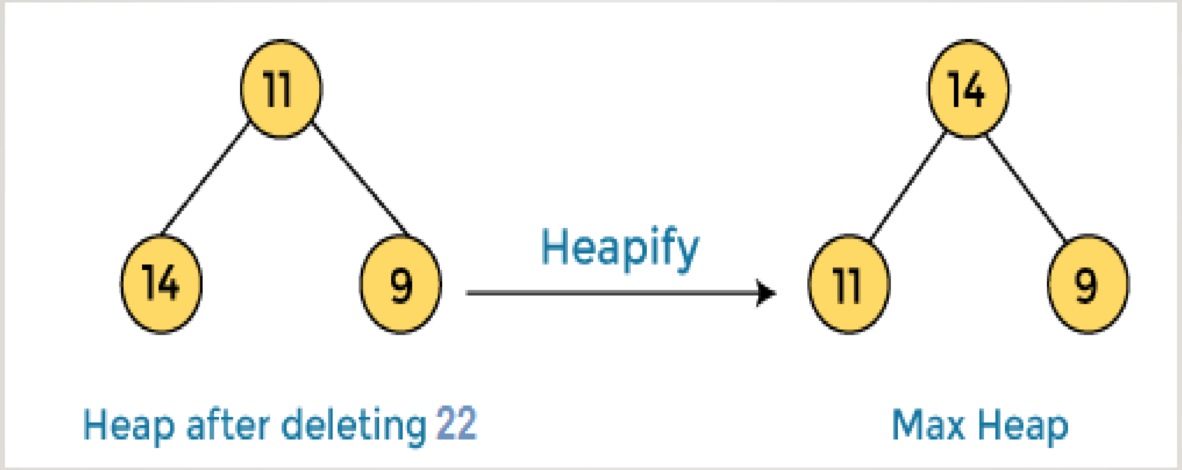


After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are



22	14	9	11	54	76	81	89

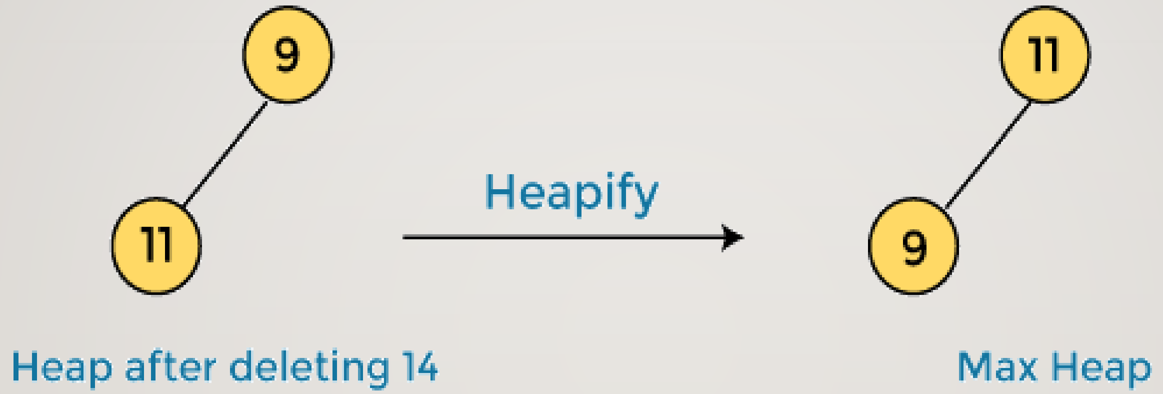
In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -

14	11	9	22	54	76	81	89

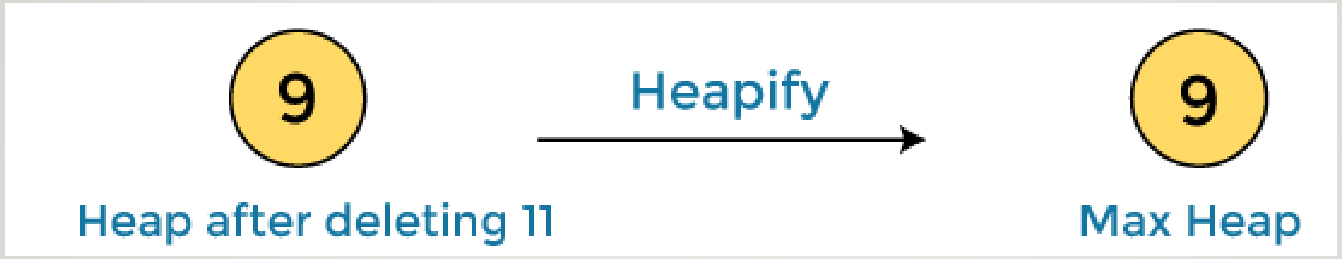
In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



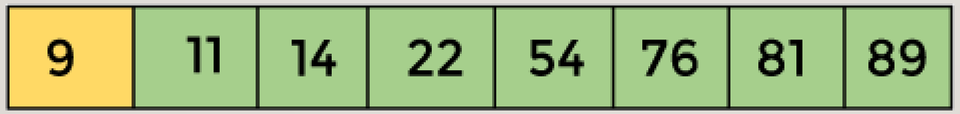
After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -

11	9	14	22	54	76	81	89

In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

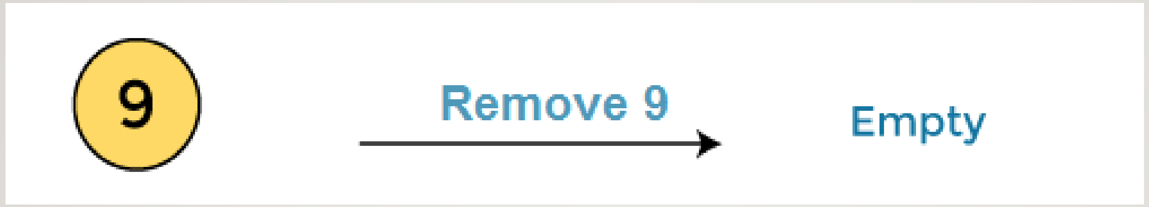


After swapping the array element **11** with **9,** the elements of array are -

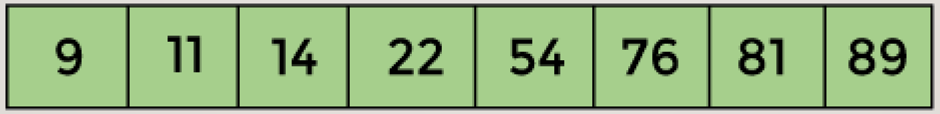


9	11	14	22	54	76	81	89

Now, heap has only one element left. After deleting it, heap will be empty.



After completion of sorting, the array elements are -



9	11	14	22	54	76	81	89

Now, the array is completely sorted.

Time Complexity

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **O(n log n).**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **O(n log n).**