Heap Sort Algorithm

In this article, we will discuss the Heapsort Algorithm. Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heap sort basically recursively performs two main operations -

- Build a heap H, using the elements of array.
- Repeatedly delete the root element of the heap formed in 1st phase.

Before knowing more about the heap sort, let's first see a brief description of **Heap.**

What is a heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

What is heap sort?

Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

Heapsort is the in-place sorting algorithm.

Now, let's see the algorithm of heap sort.

Algorithm

```
1. HeapSort(arr)
2. BuildMaxHeap(arr)
3. for i = length(arr) to 2
4.    swap arr[1] with arr[i]
5.       heap_size[arr] = heap_size[arr] ? 1
6.       MaxHeapify(arr,1)
7. End
```

BuildMaxHeap(arr)

```
1. BuildMaxHeap(arr)
2.     heap_size(arr) = length(arr)
3.     for i = length(arr)/2 to 1
4. MaxHeapify(arr,i)
5. End
```

MaxHeapify(arr,i)

```
1. MaxHeapify(arr,i)
2. L = left(i)
3. R = right(i)
4. if L ? heap_size[arr] and arr[L] > arr[i]
5. largest = L
6. else
7. largest = i
8. if R ? heap_size[arr] and arr[R] > arr[largest]
9. largest = R
10. if largest != i
11. swap arr[i] with arr[largest]
12. MaxHeapify(arr,largest)
13. End
```

Working of Heap sort Algorithm

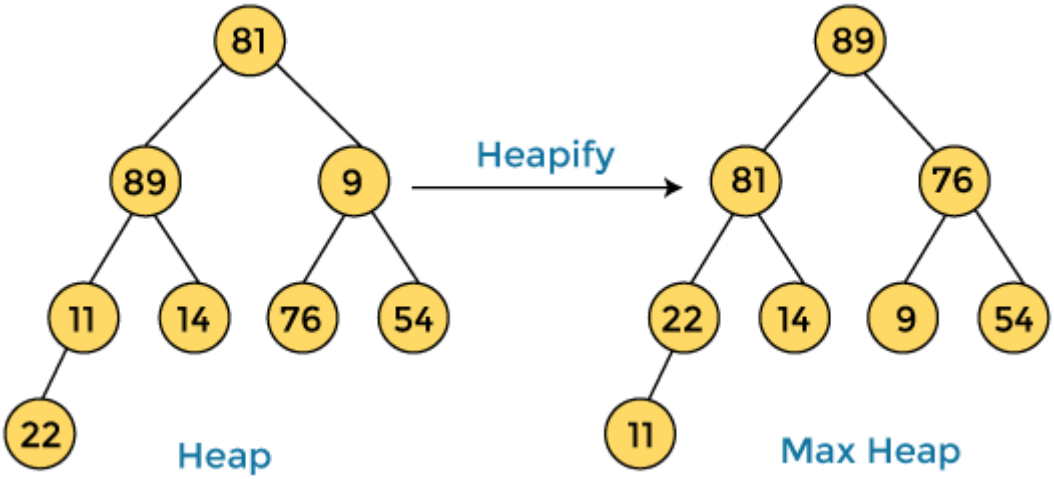
Now, let's see the working of the Heapsort Algorithm.

In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

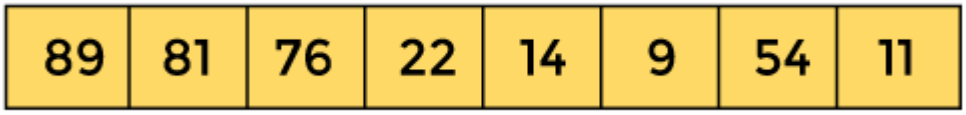
- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.

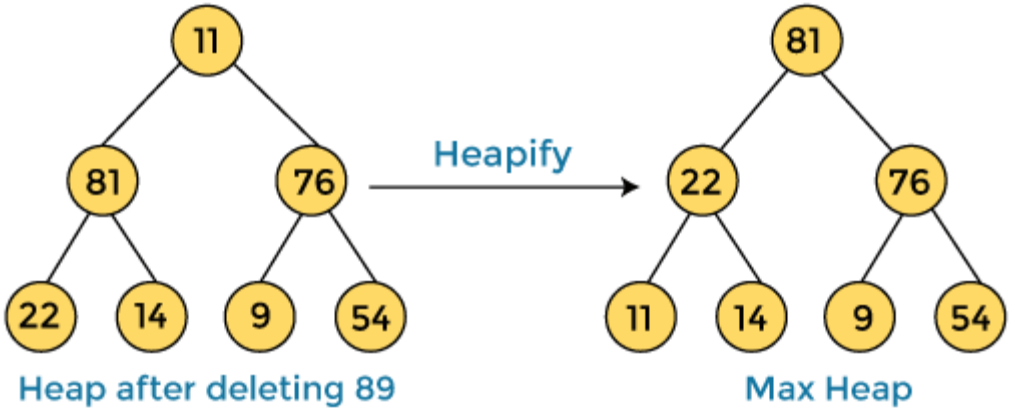
First, we have to construct a heap from the given array and convert it into max heap.



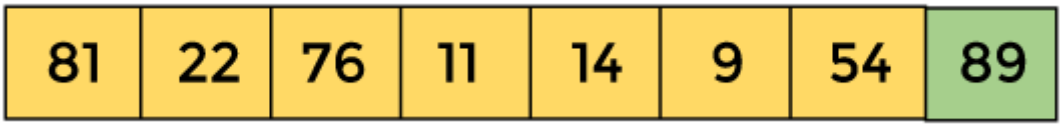
After converting the given heap into max heap, the array elements are -



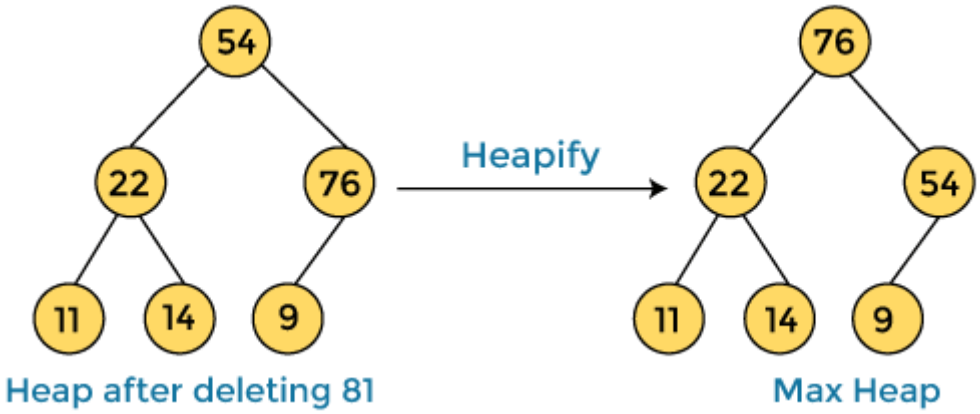
Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



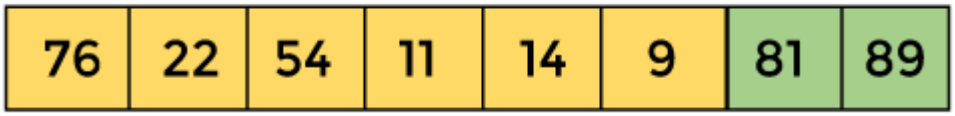
After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -



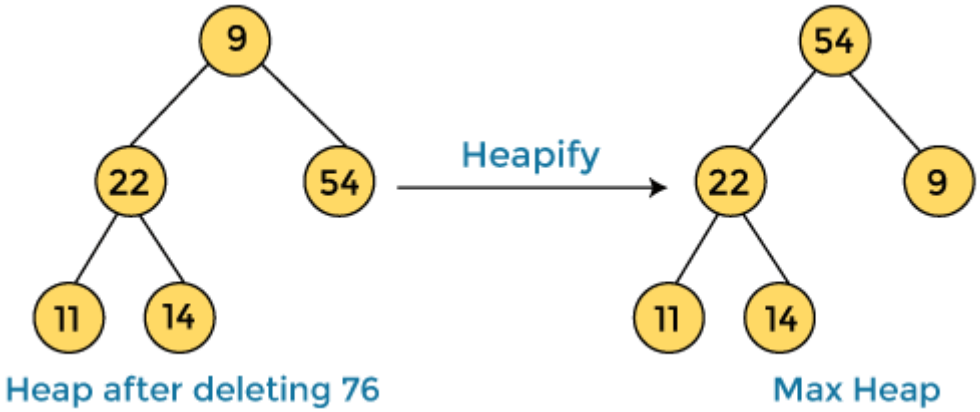
In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



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

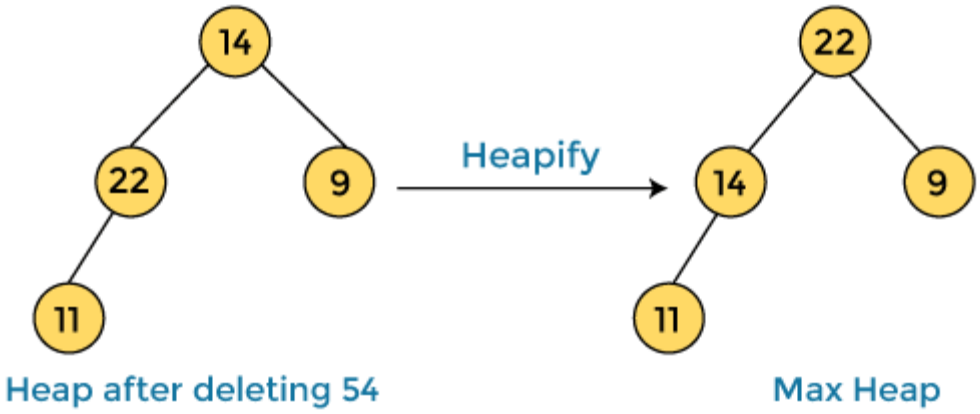


In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

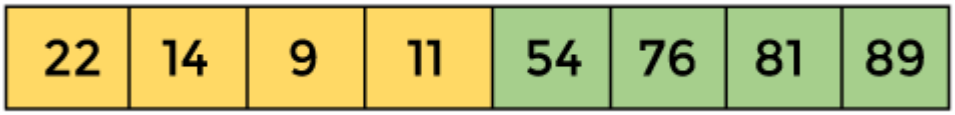


After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -

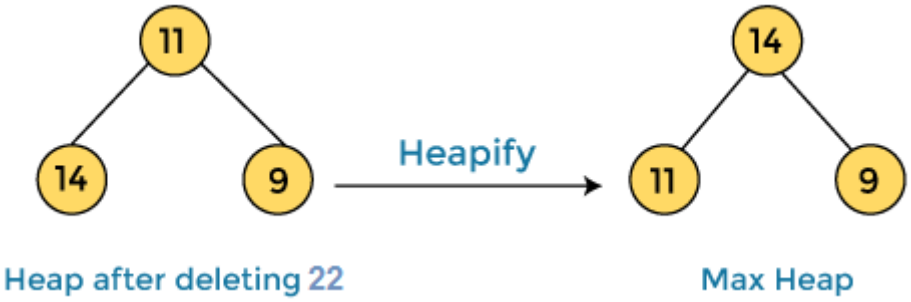
In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



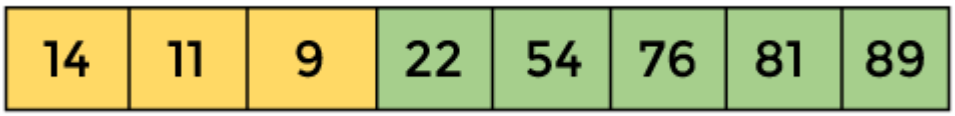
After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -



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.

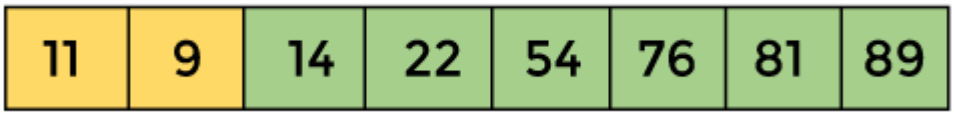


After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -



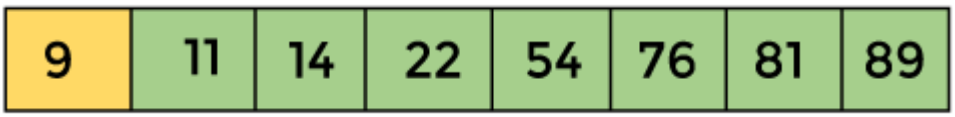
In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

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



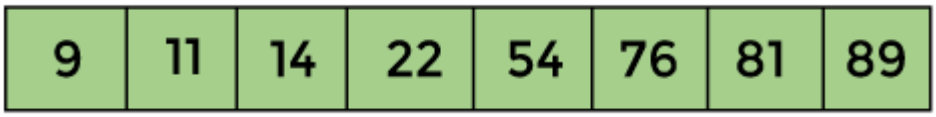
In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

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



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

After completion of sorting, the array elements are -



Now, the array is completely sorted.

Heap sort complexity

Now, let's see the time complexity of Heap sort in the best case, average case, and worst case. We will also see the space complexity of Heapsort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n logn)
Average Case	O(n log n)
Worst Case	O(n log n)

- **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).**

The time complexity of heap sort is **O(n logn)** in all three cases (best case, average case, and worst case). The height of a complete binary tree having n elements is **logn.**

2. Space Complexity

Space Complexity	O
Stable	N

- The space complexity of Heap sort is O(1).

Implementation of Heapsort

Now, let's see the programs of Heap sort in different programming languages.

Program: Write a program to implement heap sort in C language.

```
#include <stdio.h>
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of heap. */
void heapify(int a[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child
    // If left child is larger than root
    if (left < n && a[left] > a[largest])
        largest = left;
    // If right child is larger than root
    if (right < n && a[right] > a[largest])
        largest = right;
    // If root is not largest
    if (largest != i) {
        // swap a[i] with a[largest]
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;

        heapify(a, n, largest);
    }
}
/*Function to implement the heap sort*/
void heapSort(int a[], int n)
{
```

```
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(a, n, i);
    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        /* Move current root element to end*/
        // swap a[0] with a[i]
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;

        heapify(a, i, 0);
    }
}
/* function to print the array elements */
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
    {
        printf("%d", arr[i]);
        printf(" ");
    }

}
int main()
{
    int a[] = {48, 10, 23, 43, 28, 26, 1};
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    heapSort(a, n);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
    return 0;
}
```

Output

```
Before sorting array elements are -
48 10 23 43 28 26 1
After sorting array elements are -
1 10 23 26 28 43 48
```

Program: Write a program to implement heap sort in C++.

```
#include <iostream>
using namespace std;
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of heap. */
void heapify(int a[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child
    // If left child is larger than root
    if (left < n && a[left] > a[largest])
        largest = left;
    // If right child is larger than root
    if (right < n && a[right] > a[largest])
        largest = right;
    // If root is not largest
    if (largest != i) {
        // swap a[i] with a[largest]
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;

        heapify(a, n, largest);
    }
}
/*Function to implement the heap sort*/
void heapSort(int a[], int n)
{

    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(a, n, i);
    // One by one extract an element from heap
```

```
33.     for (int i = n - 1; i >= 0; i--) {
34.         /* Move current root element to end*/
35.         // swap a[0] with a[i]
36.         int temp = a[0];
37.         a[0] = a[i];
38.         a[i] = temp;
39.
40.         heapify(a, i, 0);
41.     }
42. }
43. /* function to print the array elements */
44. void printArr(int a[], int n)
45. {
46.     for (int i = 0; i < n; ++i)
47.     {
48.         cout<<a[i]<<" ";
49.     }
50.
51. }
52. int main()
53. {
54.     int a[] = {47, 9, 22, 42, 27, 25, 0};
55.     int n = sizeof(a) / sizeof(a[0]);
56.     cout<<"Before sorting array elements are - \n";
57.     printArr(a, n);
58.     heapSort(a, n);
59.     cout<<"\nAfter sorting array elements are - \n";
60.     printArr(a, n);
61.     return 0;
62. }
```

Output

```
Before sorting array elements are -
47 9 22 42 27 25 0
After sorting array elements are -
0 9 22 25 27 42 47
```

Program: Write a program to implement heap sort in C#.

```
using System;
class HeapSort {
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of heap. */
static void heapify(int[] a, int n, int i)
{
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child
    // If left child is larger than root
    if (left < n && a[left] > a[largest])
        largest = left;
    // If right child is larger than root
    if (right < n && a[right] > a[largest])
        largest = right;
    // If root is not largest
    if (largest != i) {
        // swap a[i] with a[largest]
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;

        heapify(a, n, largest);
    }
}
/*Function to implement the heap sort*/
static void heapSort(int[] a, int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(a, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        /* Move current root element to end*/
        // swap a[0] with a[i]
        int temp = a[0];
        a[0] = a[i];
```

```
38.         a[i] = temp;
39.
40.         heapify(a, i, 0);
41.     }
42. }
43. /* function to print the array elements */
44. static void printArr(int[] a, int n)
45. {
46.     for (int i = 0; i < n; ++i)
47.         Console.Write(a[i] + " ");
48. }
49. static void Main()
50. {
51.     int[] a = {46, 8, 21, 41, 26, 24, -1};
52.     int n = a.Length;
53.     Console.Write("Before sorting array elements are - \n");
54.     printArr(a, n);
55.     heapSort(a, n);
56.     Console.Write("\nAfter sorting array elements are - \n");
57.     printArr(a, n);
58. }
59. }
```

Output

```
Before sorting array elements are -
46 8 21 41 26 24 -1
After sorting array elements are -
-1 8 21 24 26 41 46
```

Program: Write a program to implement heap sort in Java.

```
1. class HeapSort
2. {
3. /* function to heapify a subtree. Here 'i' is the
4. index of root node in array a[], and 'n' is the size of heap. */
5. static void heapify(int a[], int n, int i)
6. {
7.     int largest = i; // Initialize largest as root
```

```
8.      int left = 2 * i + 1; // left child
9.      int right = 2 * i + 2; // right child
10.     // If left child is larger than root
11.     if (left < n && a[left] > a[largest])
12.         largest = left;
13.     // If right child is larger than root
14.     if (right < n && a[right] > a[largest])
15.         largest = right;
16.     // If root is not largest
17.     if (largest != i) {
18.         // swap a[i] with a[largest]
19.         int temp = a[i];
20.         a[i] = a[largest];
21.         a[largest] = temp;
22.
23.         heapify(a, n, largest);
24.     }
25. }
26. /*Function to implement the heap sort*/
27. static void heapSort(int a[], int n)
28. {
29.     for (int i = n / 2 - 1; i >= 0; i--)
30.         heapify(a, n, i);
31.
32.     // One by one extract an element from heap
33.     for (int i = n - 1; i >= 0; i--) {
34.         /* Move current root element to end*/
35.         // swap a[0] with a[i]
36.         int temp = a[0];
37.         a[0] = a[i];
38.         a[i] = temp;
39.
40.         heapify(a, i, 0);
41.     }
42. }
43. /* function to print the array elements */
44. static void printArr(int a[], int n)
```

```
45. {
46.     for (int i = 0; i < n; ++i)
47.         System.out.print(a[i] + " ");
48. }
49. public static void main(String args[])
50. {
51.     int a[] = {45, 7, 20, 40, 25, 23, -2};
52.     int n = a.length;
53.     System.out.print("Before sorting array elements are - \n");
54.     printArr(a, n);
55.     heapSort(a, n);
56.     System.out.print("\nAfter sorting array elements are - \n");
57.     printArr(a, n);
58. }
59. }
```

Output

```
D:\JTP>javac HeapSort.java

D:\JTP>java  HeapSort
Before sorting array elements are -
45 7 20 40 25 23 -2
After sorting array elements are -
-2 7 20 23 25 40 45
D:\JTP>
```