**Binary Heap**

*A****Binary Heap****is a*[***complete Binary Tree***](https://www.geeksforgeeks.org/complete-binary-tree/)*which is used to store data efficiently to get the max or min element based on its structure.*

A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at the root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

### Examples of Min Heap:

*10                       10  
         /      \                 /         \    
     20     100        15           30    
   /                        /    \         /    \  
30                     40   50   100   40*

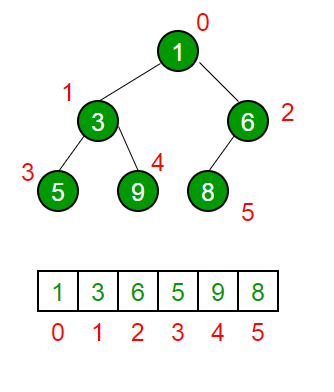
## How is Binary Heap represented?

A Binary Heap is a **Complete Binary Tree**. A binary heap is typically represented as an array.

* The root element will be at Arr[0].
* The below table shows indices of other nodes for the ith 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 |

The traversal method use to achieve Array representation is[**Level Order Traversal**](https://www.geeksforgeeks.org/level-order-tree-traversal/). Please refer to [Array Representation Of Binary Heap](https://www.geeksforgeeks.org/array-representation-of-binary-heap/) for details.



## Operations on Heap:

Below are some standard operations on min heap:

* **getMin():** It returns the root element of Min Heap. The time Complexity of this operation is **O(1)**. In case of a maxheap it would be **getMax()**.
* **extractMin():** Removes the minimum element from MinHeap. The time Complexity of this Operation is **O(log N)** as this operation needs to maintain the heap property (by calling **heapify()**) after removing the root.
* **decreaseKey():** Decreases the value of the key. The time complexity of this operation is **O(log N)**. If the decreased key value of a node is greater than the parent of the node, then we don’t need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
* **insert():** Inserting a new key takes **O(log N)** time. We add a new key at the end of the tree. If the new key is greater than its parent, then we don’t need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
* **delete():** Deleting a key also takes **O(log N)** time. We replace the key to be deleted with the minimum infinite by calling **decreaseKey()**. After decreaseKey(), the minus infinite value must reach root, so we call **extractMin()** to remove the key.

## Applications of Heaps:

* [Heap Sort](https://www.geeksforgeeks.org/heap-sort/): Heap Sort uses Binary Heap to sort an array in O(nLogn) time.
* [Priority Queue:](https://www.geeksforgeeks.org/priority-queue-set-1-introduction/) Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(log N) time. Binomial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently.
* Graph Algorithms: The priority queues are especially used in Graph Algorithms like [Dijkstra’s Shortest Path](https://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/) and[Prim’s Minimum Spanning Tree](https://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/).
* Many problems can be efficiently solved using Heaps. See following for example. a) [K’th Largest Element in an array](https://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/). b) [Sort an almost sorted array/](https://www.geeksforgeeks.org/nearly-sorted-algorithm/) c) [Merge K Sorted Arrays](https://www.geeksforgeeks.org/merge-k-sorted-arrays/).

***Heap sort****is a comparison-based sorting technique based on*[*Binary Heap*](http://www.geeksforgeeks.org/binary-heap/)*data structure. It is similar to the*[*selection sort*](http://www.geeksforgeeks.org/selection-sort/)*where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.*

## Heap Sort Algorithm

To solve the problem follow the below idea:

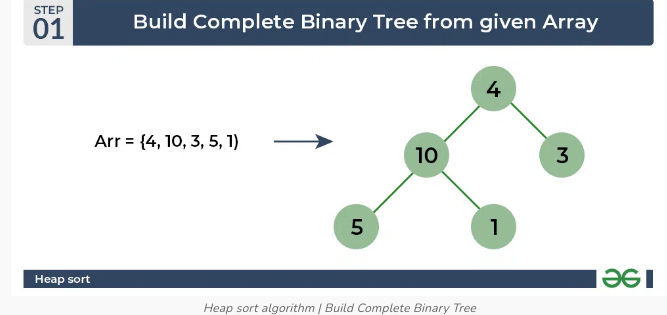
*First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.*

* *Build a heap from the given input array.*
* *Repeat the following steps until the heap contains only one element:*
  + *Swap the root element of the heap (which is the largest element) with the last element of the heap.*
  + *Remove the last element of the heap (which is now in the correct position).*
  + *Heapify the remaining elements of the heap.*
* *The sorted array is obtained by reversing the order of the elements in the input array.*

## ****Detailed Working of Heap Sort****

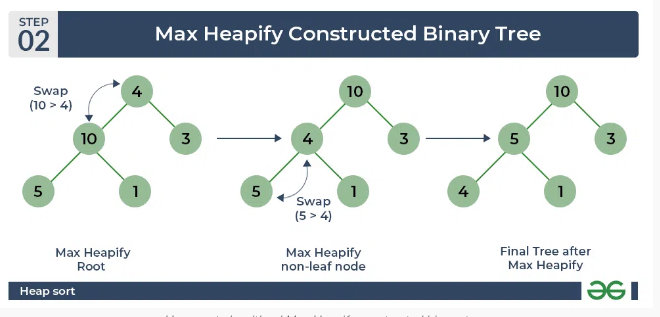
*To understand heap sort more clearly, let’s take an unsorted array and try to sort it using heap sort.  
Consider the array: arr[] = {4, 10, 3, 5, 1}.*

***Build Complete Binary Tree:****Build a complete binary tree from the array.*

**

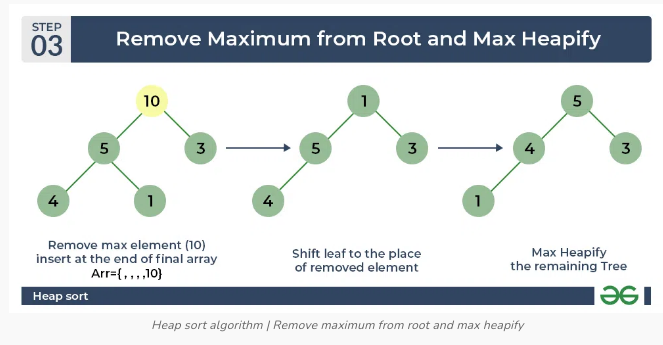
***Transform into max heap:****After that, the task is to construct a tree from that unsorted array and try to convert it into*[*max heap.*](https://www.geeksforgeeks.org/difference-between-min-heap-and-max-heap/)

* *To transform a heap into a max-heap, the parent node should always be greater than or equal to the child nodes*
  + *Here, in this example, as the parent node****4****is smaller than the child node****10,****thus, swap them to build a max-heap.*
* *Now,****4****as a parent is smaller than the child****5****, thus swap both of these again and the resulted heap and array should be like this:*

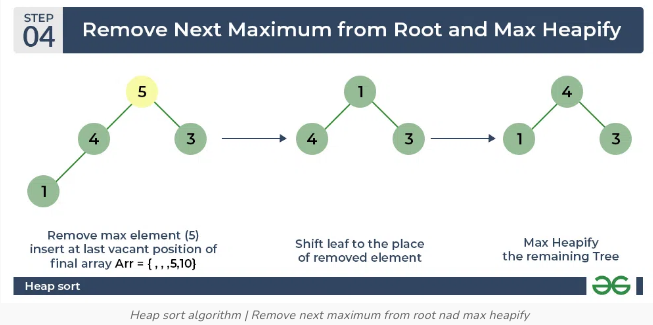
**

***Perform heap sort:****Remove the maximum element in each step (i.e., move it to the end position and remove that) and then consider the remaining elements and transform it into a max heap.*

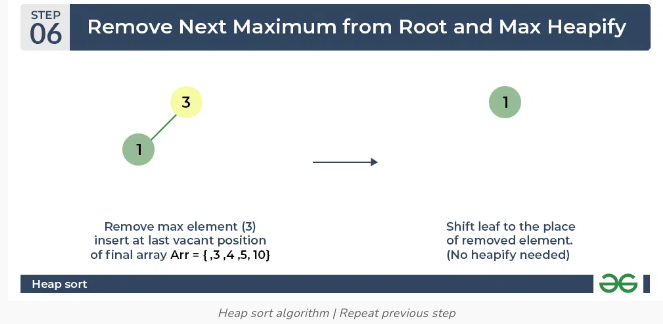
* *Delete the root element****(10)****from the max heap. In order to delete this node, try to swap it with the last node, i.e.****(1).****After removing the root element, again heapify it to convert it into max heap.*
  + *Resulted heap and array should look like this:*

**

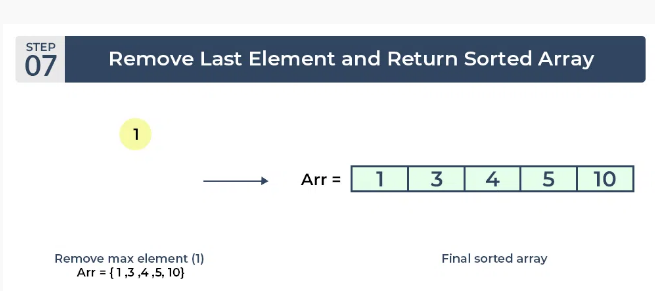
* *Repeat the above steps and it will look like the following:*

**

* *Now remove the root (i.e. 3) again and perform heapify.*



* *Now when the root is removed once again it is sorted. and the sorted array will be like****arr[] = {1, 3, 4, 5, 10}****.*



## Complexity Analysis of ****Heap Sort****

**Time Complexity:**O(n log n)  
**Auxiliary Space:**O(log n), due to the recursive call stack. However, auxiliary space can be O(1) for iterative implementation.

**Implementation of Heap Sort**

*// C++ program for implementation of Heap Sort*

#include *<iostream>*

**using** **namespace** **std**;

*// To heapify a subtree rooted with node i*

*// which is an index in arr[].*

*// n is size of heap*

void heapify(int arr[], int N, int i)

{

*// Initialize largest as root*

int largest = i;

*// left = 2\*i + 1*

int l = 2 \* i + 1;

*// right = 2\*i + 2*

int r = 2 \* i + 2;

*// If left child is larger than root*

**if** (l < N && arr[l] > arr[largest])

largest = l;

*// If right child is larger than largest*

*// so far*

**if** (r < N && arr[r] > arr[largest])

largest = r;

*// If largest is not root*

**if** (largest != i) {

swap(arr[i], arr[largest]);

*// Recursively heapify the affected*

*// sub-tree*

heapify(arr, N, largest);

}

}

*// Main function to do heap sort*

void heapSort(int arr[], int N)

{

*// Build heap (rearrange array)*

**for** (int i = N / 2 - 1; i >= 0; i--)

heapify(arr, N, i);

*// One by one extract an element*

*// from heap*

**for** (int i = N - 1; i > 0; i--) {

*// Move current root to end*

swap(arr[0], arr[i]);

*// call max heapify on the reduced heap*

heapify(arr, i, 0);

}

}

*// A utility function to print array of size n*

void printArray(int arr[], int N)

{

**for** (int i = 0; i < N; ++i)

cout << arr[i] << " ";

cout << "**\n**";

}

*// Driver's code*

int main()

{

int arr[] = { 12, 11, 13, 5, 6, 7 };

int N = **sizeof**(arr) / **sizeof**(arr[0]);

*// Function call*

heapSort(arr, N);

cout << "Sorted array is **\n**";

printArray(arr, N);

}

**Output**

Sorted array is

5 6 7 11 12 13

**Huffman Coding**

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.   
The variable-length codes assigned to input characters are [Prefix Codes](http://en.wikipedia.org/wiki/Prefix_code), means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.   
Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.  
See [this](http://en.wikipedia.org/wiki/Huffman_coding#Applications)for applications of Huffman Coding.   
There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

***Algorithm:***

The method which is used to construct optimal prefix code is called **Huffman coding**.

 This algorithm builds a tree in bottom up manner. We can denote this tree by **T**

Let, |c| be number of leaves

|c| -1 are number of operations required to merge the nodes. Q be the priority queue which can be used while constructing binary heap.

*Algorithm Huffman (c)*

*{*

*n= |c|*

*Q = c*

*for i<-1 to n-1*

*do*

*{*

*temp <- get node ()*

*left (temp] Get\_min (Q) right [temp] Get Min (Q)*

*a = left [templ b = right [temp]*

*F [temp]<- f[a] + [b]*

*insert (Q, temp)*

*}*

*return Get\_min (0)*

*}*

***Steps to build Huffman Tree***  
Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.  
   Let us understand the algorithm with an example:

character Frequency

a 5

b 9

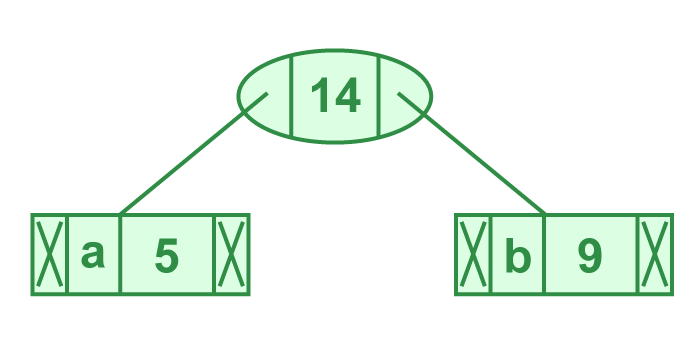
c 12

d 13

e 16

f 45

**Step 1.** Build a min heap that contains 6 nodes where each node represents root of a tree with single node.  
**Step 2** Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14. 



*Illustration of step 2*

Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character Frequency

c 12

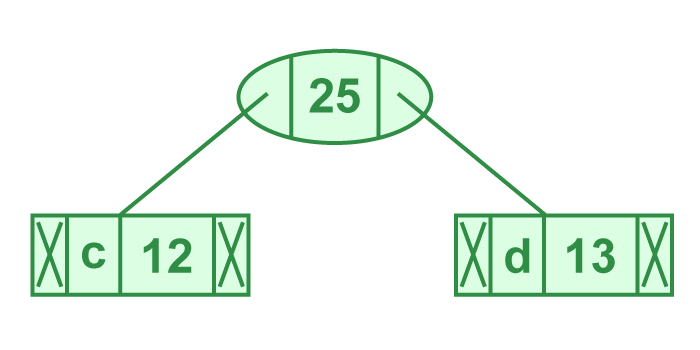
d 13

Internal Node 14

e 16

f 45

**Step 3:** Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



*Illustration of step 3*

Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

character Frequency

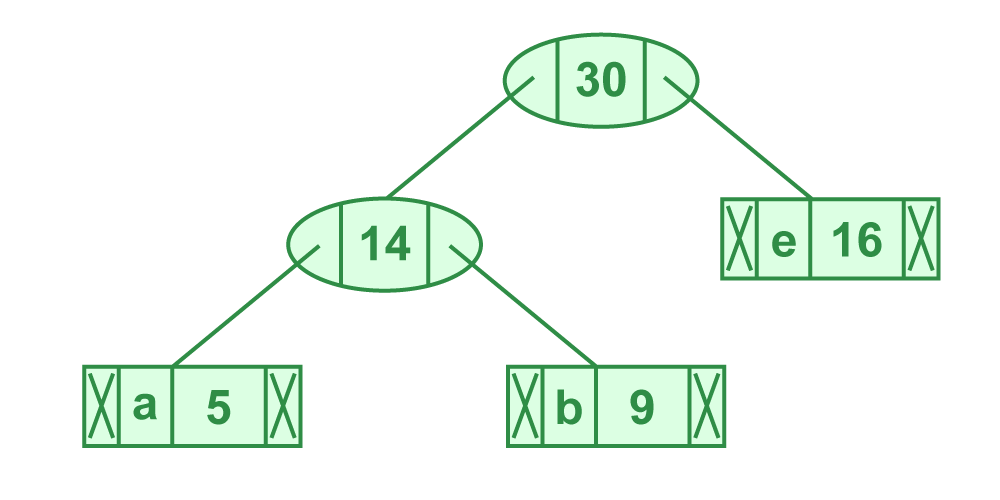
Internal Node 14

e 16

Internal Node 25

f 45

**Step 4:** Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



*Illustration of step 4*

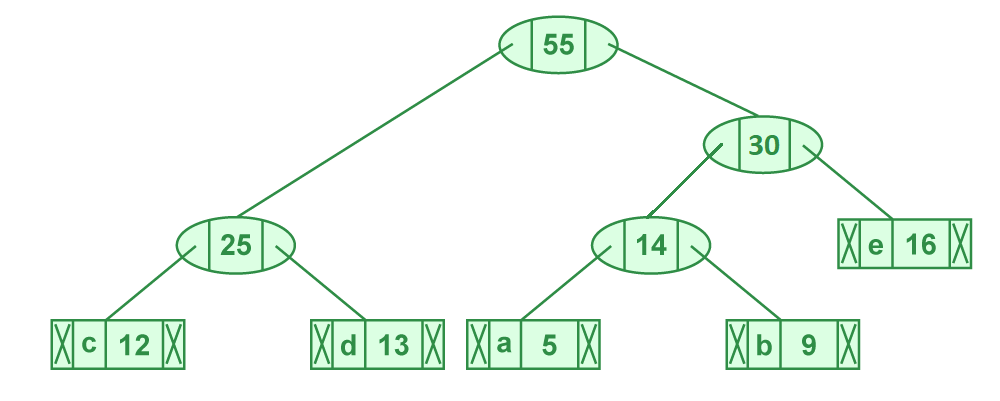
Now min heap contains 3 nodes.

character Frequency

Internal Node 25

Internal Node 30

f 45

**Step 5:** Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55

*Illustration of step 5*

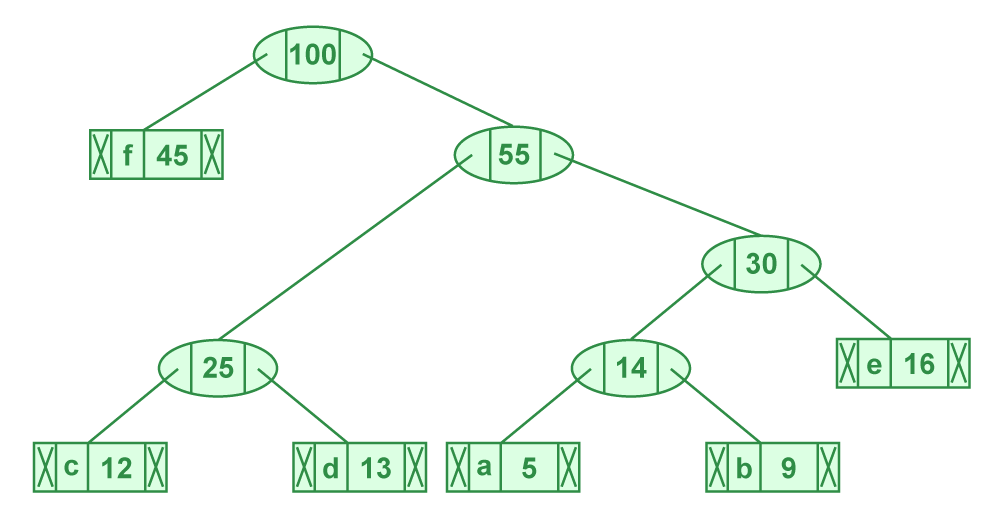
Now min heap contains 2 nodes.

character Frequency

f 45

Internal Node 55

**Step 6:** Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



*Illustration of step 6*

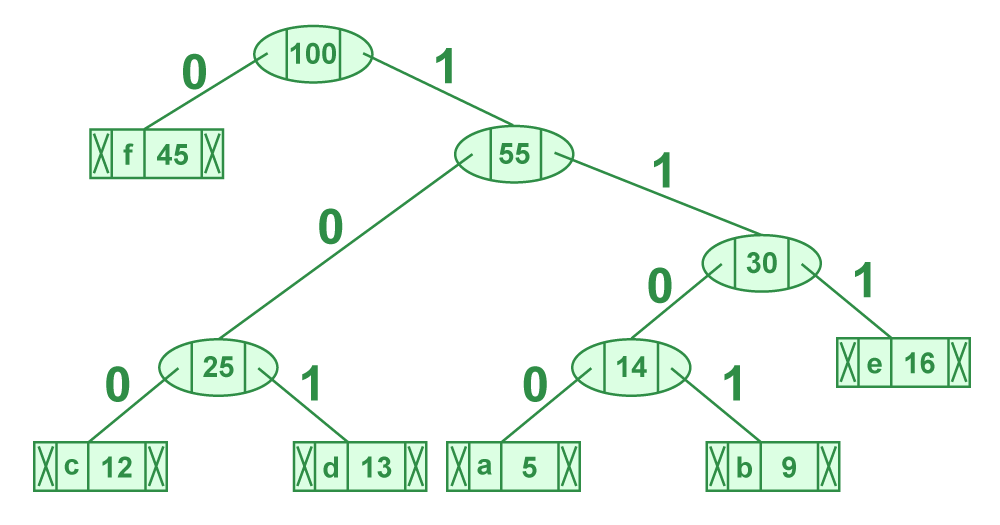
Now min heap contains only one node.

character Frequency

Internal Node 100

Since the heap contains only one node, the algorithm stops here.

***Steps to print codes from Huffman Tree:***  
Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



*Steps to print code from HuffmanTree*

The codes are as follows:

character code-word

f 0

c 100

d 101

a 1100

b 1101

e 111