

[Home](#) › [Learn](#) › [Heap Sort Algorithm: Explanation, Implementation, and Complexity](#)

Heap Sort Algorithm: Explanation, Implementation, and Complexity

Whether it is for the software developer, coding engineer, software engineer, or any such position in the IT industry, heap sort is an essential part of the technical interview prep. It's almost as if its primary use is cracking job interviews! It is rarely used in real-world scenarios, despite being one of the most interesting sorting algorithms.

If you are preparing for a tech interview, check out our [technical interview checklist](#), [interview questions](#) page, and [salary negotiation e-book](#) to get interview-ready!

Having trained **over 13,500 software engineers**, we know what it takes to crack the toughest tech interviews. Our alums consistently land offers from FAANG+ companies. The highest-ever offer received by an IK alum is a whopping **\$1.267 Million!**

At IK, you get the unique opportunity to learn from **expert instructors** who are **hiring managers and tech leads** at Google, Facebook, Apple, and other top Silicon Valley tech companies.

*Want to nail your next tech interview? Sign up for our **FREE Webinar**.*

This article will discuss the following:

- What Is Heap Sort?
- Binary Heap
- Heapify Method
- Applications of Heap Sort
- How Does Heap Sort Work?
- Heap Sort Algorithm
- Heap Sort Pseudocode
- Heap Sort Code
- Heap Sort Complexities
- Strengths and Weaknesses of Heap Sort
- FAQs on Heap Sort

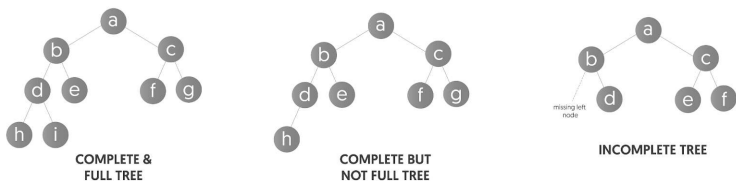
What Is Heap Sort?

To understand how heap sort works, we first need to understand some basic concepts related to binary heaps. Feel free to skip them if you are already familiar with these concepts.

Binary Heap

Heap is a tree-based data structure in which all the tree nodes are in a particular order, such that the tree satisfies the heap properties (that is, *a specific parent-child relationship* is followed throughout the tree).

A heap data structure where the tree is a complete binary tree is referred to as a binary heap.



A **complete binary tree** is a binary tree in which:

- All levels except the bottom-most level are completely filled.
- All nodes in the bottom-most level are as far left as possible.
- The last level may or may not be completely filled.

A **full binary tree** is a binary tree where every node has 0 or 2 children.

Properties of a Binary Heap

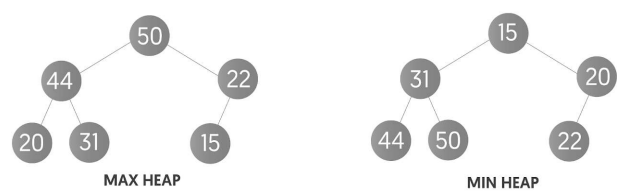
1. They are complete binary trees: This means all levels are totally filled (except maybe the last level), and the nodes in the last level are as left as possible. This property makes arrays a suitable data structure for storing binary heaps.

We can easily calculate the indices of a node's children. So, for *parent index i*, the *left child* will be found at *index 2*i+1*, and the *right child* will be found at *index 2*i+2* (for indices that start with 0). Similarly, for a child at index *i*, its parent can be found at *index floor((i-1)/2)*.



2. Heaps are mainly of two types — max heap and min heap:

- In a *max heap*, the *value of a node* is always \geq the *value of each of its children*.
- In a *min heap*, the *value of a parent* is always \leq the *value of each of its children*.



3. Root element: In a max heap, the element at the root will always be the maximum. In a min heap, the root element will always be the smallest. The heap sort algorithm takes advantage of this property to sort an array using heaps.

Heap Sort Definition

Heap sort is an efficient comparison-based sorting algorithm that:

- Creates a heap from the input array.
- Then sorts the array by taking advantage of a heap's properties.

Heapify Method

Before going into the workings of heap sort, we'll visualize the array as a complete binary tree. Next, we turn it into a max heap using a process called **heapification**.

The brilliance of heapification lies in the following fact:

If all the subtrees in a binary tree are MaxHeaps themselves, the whole tree is a MaxHeap.

One way to implement this idea would be:

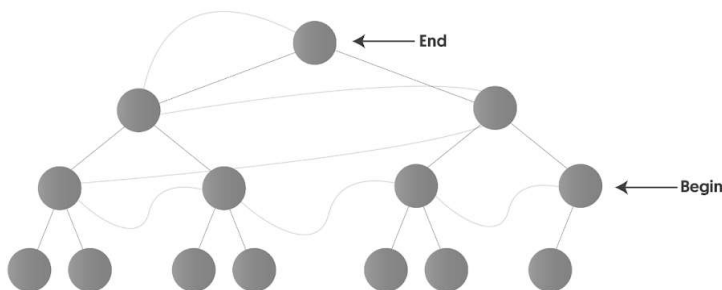
1. Start at the bottom of the tree.
2. Iterate through all the nodes as we travel to the top.
3. At each step, ensure that the node and all its children form a valid max heap.

If we successfully do that, we will have transformed the whole binary tree into a valid MaxHeap after processing all the nodes.

One way to optimize this process is by ignoring all the *leaf nodes* since they don't have any children:

1. Go to the **right-most node** in the **second bottom-most layer**, which has any **children**.
2. Process that right-most node to make sure it **forms a MaxHeap with its children**
3. Traverse to the node to its **left** and repeat the process.
4. At the end of the level, we jump to the right-most node of the level above it.

This journey ends when we eventually reach the topmost node and process it.



1. Compare the *value of the node* with the *value of the child nodes*. If the *parent's value* is more than each of the *values of the child nodes*, do nothing.
2. If the *value of a child node* is more than *the parent node*, *swap the values* between the parent and the child node. (If both the child nodes have a higher value than the parent, swap the parent's value with the value of the child who has the greater value of the two children.)
3. Now, for the child node that got updated, we repeat steps 1 and 2.

Recursion:

If this sounds like a recursive method, that's because it is! We keep calling this method recursively for the child nodes that got updated until we reach a stage where the child node is either a leaf or has children, each of whose values are lower.

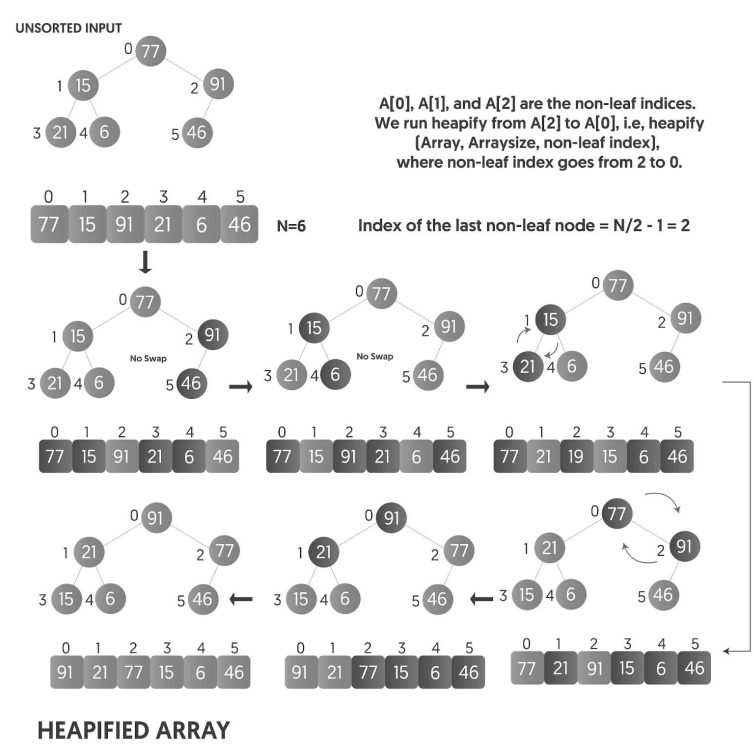
Bottom-to-top traversal:

You might have wondered why we decided to traverse bottom to top and not top to bottom. That's because steps 1-3 for heapifying a node work only if the child nodes are heapified already.

Max/Min heap formation:

At the end of this process, a max heap is fully formed. We can also make a *min heap* simply by changing the condition to “*parent value should be <= each of its children's values*” (swap values if the condition isn't met).

Look at the following example:



- When sorting *in-place*, we can use a max heap to sort the array in ascending order and a min heap to sort the array in descending order.
- If sorting doesn't have to be *in-place*, we can use an auxiliary array to place the extracted element from the heap's top in its correct position, whether we use a min heap or a max heap for the sorting.

But even when sorting is not the aim, a min/max heap in itself is a useful construction:

- The root element of a max heap always contains the maximum element.
- The root element of a min heap always has the minimum element.

This quality of heaps can come in handy when we want to extract only the largest or smallest element from an array without sorting the remaining elements.

Applications of Heap Sort

Heap sort has limited usage since algorithms like **merge sort** and **quicksort** are better in practice. We extensively use heaps for problems like getting the largest or smallest elements in an array, sorting an almost sorted array, etc.

Some key applications of Heap sort include:

- Implementation of priority queues
- Security systems
- Embedded systems (for example, Linux Kernel)

How Does Heap Sort Work?

Now that we’ve learned how to create a heap from an array using the heapify method, we will look into using the heap to sort the array.

After the *heap formation using the heapify method*, the sorting is done by:

1. **Swapping** the *root element* with the *last element of the array* and decreasing the length of the heap array by one. (In heap representation, it is equivalent to swapping the root with the bottom-most and right-most leaf and then deleting the leaf.)
 2. **Restoring** heap properties (*reheapification*) *after each deletion*, where we need to apply the heapify method *only on the root node*. The subtree heaps will still have their heap properties intact at the beginning of the process.
 3. **Repeating** this process until every element in the array is sorted: Root removal, its storage *in the position of the highest index value used by the heap*, and heap length decrement.
- On a **max heap**, this process will sort the array in **ascending order**.
 - On a **min heap**, this process will sort in **descending order**.

This process can be best illustrated using an example:



The process above ends when heap size = 2 because a two-element heap is always considered sorted.

So basically, the heap sort algorithm has two parts that run recursively till heap size ≥ 2 :

- **Creating a heap** from the currently unsorted elements of the array.
- **Swapping the root element with the last element** of the heap (right-most leaf node)
- **Reducing heap size** by 1.

Heap Sort Algorithm

Here’s the algorithm for heap sort:

Step 1: Build Heap. Build a heap from the input data. Build a max heap to sort in increasing order, and build a min heap to sort in decreasing order.

Step 2: Swap Root. Swap the root element with the last item of the heap.

Step 3: Reduce Heap Size. Reduce the heap size by 1.

Step 4: Re-Heapify. Heapify the remaining elements into a heap of the new heap size by calling heapify on the root node.

Step 5: Call Recursively. Repeat steps 2,3,4 as long as the heap size is greater than 2.

Each time, the *last array position* is discarded from the heap once it contains the correct element. The process is repeated until all the input array elements are sorted. This happens when the heap size is reduced to 2 since the first two elements will automatically be in order for a heap that satisfies the heap property.

Heap Sort Pseudocode

Following is the pseudocode for heap sort. Please look and try to implement this in a programming language of your choice.

Input:

```
Array A, size N
heapSort()
For all non-leaf elements (i=N/2-1;i>=0;i--)
    Build Heap (Heapify)
Initialize indexEnd
While indexEnd>1
    Swap(A[0],A[indexEnd])
    indexEnd=indexEnd-1
    Build heap (apply heapify on the root node), considering array from A[0] to A[indexEnd]
Output the sorted array[]
end heapSort()
```

Heap Sort Code

We have implemented the heap sort algorithm to sort in ascending order in C++:

The Heap Sort Program

```
#include
using namespace std;

void heapify(int array[], int sizeHeap, int parentIndex)
{
    // Establishing a relationship between indices of a node and indices of
    // its left and right children
    int larger = parentIndex;
    int leftChildIndex = 2 * parentIndex + 1;
    int rightChildIndex = 2 * parentIndex + 2;

    // Making sure the parent is greater than or equal to its left and right
    // children
    if (leftChildIndex < sizeHeap && array[leftChildIndex] > array[larger])
        larger = leftChildIndex;

    if (rightChildIndex < sizeHeap && array[rightChildIndex] > array[larger])
```

```
        larger = rightChildIndex;

// Swap and heapify if parent/root is not the largest
if (larger != parentIndex)
{
    swap(array[parentIndex], array[larger]);
    heapify(array, sizeHeap, larger);
}
}

void heapSort(int array[], int sizeArray)
{

// Creating max heap, iterating for all non=leaf indices, since leaf
// indices don't have children to check for

for (int nonleafNodeIndex = sizeArray / 2 - 1; nonleafNodeIndex >= 0; nonleafNodeIndex--)
    heapify(array, sizeArray, nonleafNodeIndex);

// Swap the root element of the heap with the last heap index, the
// Reduce heap size till it becomes 2 (last heap index
// is 1)

for (int lastHeapIndex = sizeArray - 1; lastHeapIndex >= 1; lastHeapIndex--)
{
    swap(array[0], array[lastHeapIndex]);

// Heapifying root element so that the highest element is again at the
// root
    heapify(array, lastHeapIndex, 0);
}

}

int main()
{
    int array[] = {77, 15, 91, 21, 6, 46};

    int sizeArray = sizeof(array) / sizeof(array[0]);

    heapSort(array, sizeArray);

    for (int i = 0; i < sizeArray; ++i)
        cout << array[i] << " ";
}
```

Output:



NEXT WEBINAR STARTS IN

00 : 18 : 52 : 32

DAY HRS MINS SEC

[Register for Webinar](#)

WEBINAR + LIVE Q&A

Worried about failing tech interviews?

Attend our webinar on "How to nail your next tech interview" and learn

The time complexity of heap sort is non-quadratic and comes out the same in the best, worst and average cases:

O(nlogn)

Let’s see how.

(**Note:** The following sections are based on working with MaxHeaps)

Time Complexity of

The **heapify** method is **only called on a node that is not already heapified.**

Worst-case:

- The worst-case run time will be experienced when the heapify method is run on a node smaller than all of its children. This means the node has to be swapped with its largest child. **Our tried & tested strategy for cracking interviews**
- Thus, the worst-case time complexity of each heapify method invocation is $O(h)$. **How FAANG hiring process works**
- This height is not a constant. At the bottom of the tree, h is 0, and at the top, h is equal to $\log_2 N$. **The 4 areas you must prepare for**

Register for Webinar

The time complexity for calling the heapify method for all the tree nodes (from bottom to top):

$$T(n) = \sum_{h=0}^{\lg(N)} O(h) * (\text{no. of nodes at height } h)$$



Using the fact that a heap of size N has at most $\frac{N}{2^{h+1}}$ nodes at height h , the equation becomes:

$$\begin{aligned} T(n) &= \sum_{h=0}^{\lg(N)} O(h) * \frac{N}{2^{h+1}} \\ &= O(N * \sum_{h=0}^{\lg(N)} \frac{h}{2^{h+1}}) \\ &= O(N * \sum_{h=0}^{\lg(N)} \frac{h}{2 * 2^h}) \\ &= O(\frac{N}{2} * \sum_{h=0}^{\lg(N)} \frac{h}{2^h}) \\ &= O(\frac{N}{2} * \sum_{h=0}^{\infty} \frac{h}{2^h}) \end{aligned}$$

- Taking advantage of the properties of Big-O notation, in the last step, we raised the upper limit of the summation from $\lg(N)$ to ∞ . This will help us simplify the calculation. We’ll do so with the help of known mathematical properties involving the summation of numeric expressions from 0 to ∞ .
- We will use the following mathematical property:

$$\sum_{h=0}^{\infty} h * x^h = \frac{x}{(1-x)^2}$$

- We can notice that in our equation, we can use the above property by replacing x with $1/2$. So, our equation now becomes:

$$\begin{aligned} T(n) &= O(\frac{N}{2} * \frac{\frac{1}{2}}{(1-\frac{1}{2})^2}) \\ &= O(\frac{N}{2} * 2) \\ &= O(N) \end{aligned}$$

- Thus, the first step of heap sort, which is building a heap out of a randomly arranged array, can be done in **$O(N)$** .

Time Complexity of Getting a Sorted Array Out of A max Heap

This step involves **swapping** the *left-most value* in the array with the *right-most value* in the array occupied by the heap and **reheapification** of the new smaller heap.

- **Swapping:** Swapping the max element with the bottom level right-most element and reducing the heap size can be done in constant time, **$O(1)$** .
- **Reheapification:** In the worst case, the new value at the root position will have to be swapped $\log(N)$ times to be sent to the bottom of the heap to achieve a MaxHeap once again. So each reheapification after the extraction costs **$O(\log N)$** .

We will perform this extraction N times, so the total time complexity of getting a sorted array out of a MaxHeap is **$O(N \cdot \log(N))$** .

Total Time Complexity of Heap Sort

We can calculate the total time complexity of heap sort as:

Time for creating a MaxHeap + Time for getting a sorted array out of a MaxHeap

= **$O(N) + O(N \log(N))$**

= **$O(N \log(N))$**

Heap Sort Space Complexity

Heap sort's space complexity is a constant **$O(1)$** due to its auxiliary storage.

Strengths of Heap Sort

- No quadratic worst-case run time.
- It is an in-place sorting algorithm and performs sorting in **$O(1)$** space complexity.
- Compared to quicksort, it has a better worst-case time complexity — **$O(n \log n)$** . The best-case complexity is the same for both quick sort and heap sort — **$O(n \log n)$** .
- Unlike merge sort, it does not require extra space.
- The input data being completely or almost sorted doesn't make the complexities suffer.
- The average-case complexity is the same as that of merge sort and quicksort.

Weaknesses of Heap Sort

- Heap sort is typically not stable since the operations on the heap can change the relative order of equal key items. It's typically an unstable sorting algorithm.
- If the input array is huge and doesn't fit into the memory and partitioning the array is faster than maintaining the heap, heap sort isn't an option. In such cases, something like merge sort or bucket sort, where parts of the array can be processed separately and parallelly, works best.

Heap Sort FAQs

Question 1: Does the heap data structure have to be binary-tree-based?

No, a heap does not always need to be a binary tree. But in heap sort, we use arrays to represent the heap. Using the array, we can easily calculate and track the relationship between a parent index, its left child index, and the right child index for a binary heap. And a binary heap has to be binary-tree-based.

Question 2: Can heap sort be made stable?

While heap sort is typically not stable, it can be made stable by considering the position of the elements with the same value. During heapification, treat the element towards the right as greater than the element towards the left, and your sorting will be stable.

Question 3: Why are arrays used to visualize and implement binary heaps?

Storing and accessing values in an array is faster and less complicated than using a more complex data structure. One of the main advantages of using more complex data structures is the use of methods provided by the standard library for common operations related to the data structure, e.g., push() and pop() methods for a stack.

However, storing a complete binary tree in an array still allows us to perform all operations relevant to the tree with much ease. We can find the left child, right child, parent node, root, and the last element of a tree with basic arithmetic operations on the index of the current node or the variable maintaining the size of the tree.

Question 4: How much time does it take to find the maximum and minimum element in a max heap?

The maximum element is present at the root and can be found in **O(1)** time. The minimum element will be present in the leaf nodes, and all leaf nodes have to be checked to find the minimum element. Hence, the minimum element can be found in **O(n)** time.

Question 5: What is heap sort’s space complexity and why?

Heap sort’s space complexity is a constant **O(1)** due to its auxiliary storage.

Ready to Nail Your Next Coding Interview?

Whether you’re a coding engineer gunning for a software developer or software engineer role, a tech lead, or targeting management positions at top companies, IK offers courses specifically designed for your needs to help you with your technical interview preparation!

If you’re looking for guidance and help with getting started, **sign up for our FREE webinar**. As pioneers in technical interview preparation, we have trained thousands of software engineers to crack the most challenging coding interviews and land jobs at their dream companies, such as Google, Facebook, Apple, Netflix, Amazon, and more!

Sign up now!

WEBINAR +LIVE Q&A

Worried about failing tech interviews?





Attend our webinar on

"How to nail your next tech interview" and learn



Hosted By
Ryan Valles

Founder, Interview Kickstart

- ✓  **Our tried & tested strategy for cracking interviews**
- ✓  **How FAANG hiring process works**
- ✓  **The 4 areas you must prepare for**
- ✓  **How you can accelerate your learnings**

[Register for Webinar](#)

Recommended Posts

**Sed Command in
Linux/Unix With
Examples**

**C++ STL Container
Fundamentals: Set**

**C++ STL Container
Fundamentals: Stack**

[All Posts](#)

Ready to Enroll?

Get your enrollment process started by registering for a Pre-enrollment Webinar with one of our Founders.

[Register for our Webinar](#)

NEXT WEBINAR STARTS IN

00 : 18 : 52 : 32
DAY HRS MINS SEC



[About us](#) [Why us](#) [Reviews](#) [Product](#) [Cost](#) [FAQ](#) [Blog](#) [Instructors](#) [Curriculum](#) [Careers](#) [Contact](#)
[Interview Questions](#) [Companies](#) [Learn](#) [Problems](#) [Career Advice](#)

