# What is a Heap

A brief introduction to Heaps and their uses. We will also look at Heap Property and how a Heap is represented on an array.

**We'll cover the following**

* + [Introduction](https://www.educative.io/courses/data-structures-coding-interviews-java/g75kWBBLxPG#Introduction)
    - [🔍 What is a Binary Heap?](https://www.educative.io/courses/data-structures-coding-interviews-java/g75kWBBLxPG#%F0%9F%94%8D-What-is-a-Binary-Heap?)
    - [1. Complete Binary Tree](https://www.educative.io/courses/data-structures-coding-interviews-java/g75kWBBLxPG#1.-Complete-Binary-Tree)
    - [2. Heap Property](https://www.educative.io/courses/data-structures-coding-interviews-java/g75kWBBLxPG#2.-Heap-Property)
      * [2.1 Max Heap Property:](https://www.educative.io/courses/data-structures-coding-interviews-java/g75kWBBLxPG#2.1-Max-Heap-Property:)
      * [2.2 Min Heap Property:](https://www.educative.io/courses/data-structures-coding-interviews-java/g75kWBBLxPG#2.2-Min-Heap-Property:)

## Introduction[#](https://www.educative.io/courses/data-structures-coding-interviews-java/g75kWBBLxPG#Introduction)

Heaps are advanced data structures based on Binary Trees, which is why they are commonly known as **Binary Heaps**.

### 🔍 What is a Binary Heap?[#](https://www.educative.io/courses/data-structures-coding-interviews-java/g75kWBBLxPG#%F0%9F%94%8D-What-is-a-Binary-Heap?)

A Binary Heap is a complete Binary Tree which satisfies the Heap ordering property.

Heaps are implemented through Arrays, where each element of the array corresponds to a node in the binary tree and the value inside the node is called a “key”. Heaps can also be implemented using trees with a node class and pointers, but it’s usually easier and more space efficient to use an array.

All the nodes are ordered according to the rules listed below:

1. A Heap tree must be a Complete Binary Tree.
2. The nodes must be ordered according to the Heap Property.

**Common Misconception**

There is also a common misconception that the elements of a Heap are sorted. They are not at all sorted; in fact, the only key feature of a Heap is that the largest or smallest element is always placed at the top (parent node) depending on what kind of Heap we are using.

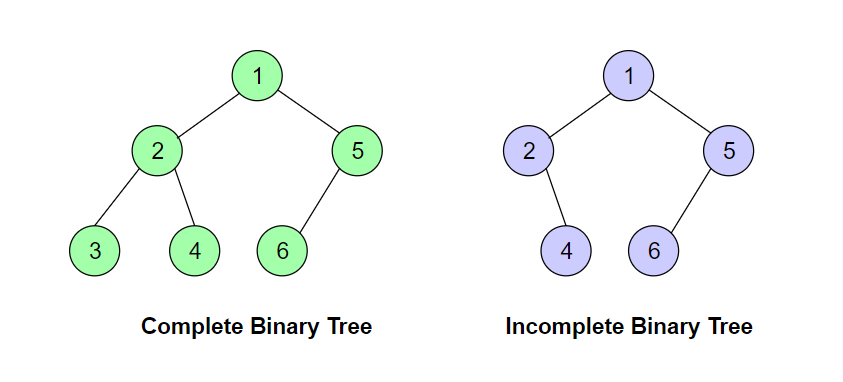
Moreover, this Data Structure Heap has nothing to do with the dynamic memory allocations on a Heap in various languages like C/C++ and Pascal.

### 1. Complete Binary Tree[#](https://www.educative.io/courses/data-structures-coding-interviews-java/g75kWBBLxPG#1.-Complete-Binary-Tree)

As discussed in the previous chapter, a Complete Binary Tree is a tree where each node has a max. of two children and nodes at all levels are completely filled (except the leaf nodes). But the nodes at the last level must be structured in such a way that the left side is never empty. This is the only condition that differentiates Complete Binary Trees from other trees.

The new elements are inserted from left to right. When you add a new node, you must make sure that the left child of that intermediate parent node is filled. If it’s not, add a node at the left and insert the new element there.

A Heap uses Complete Binary Trees to avoid holes in the array. See the figure below to see the difference between that and an Incomplete Binary Tree:



### 2. Heap Property[#](https://www.educative.io/courses/data-structures-coding-interviews-java/g75kWBBLxPG#2.-Heap-Property)

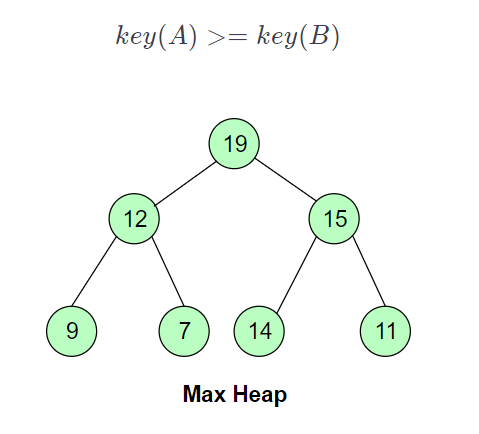
A heap is built, based on the Heap property, by comparing the parent node key with its child node keys. This comparison is done based on the heap property. The two heap structures that we are going to cover in this chapter are:

* Min Heap
* Max Heap

Min Heap is built on the Min Heap property, and Max Heap is implemented on the Max Heap property. Let’s see how they are different.

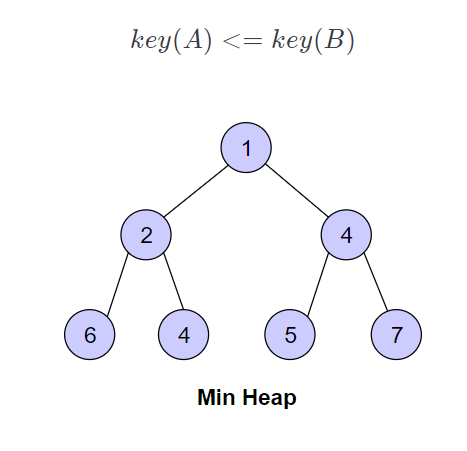
#### 2.1 Max Heap Property:[**#**](https://www.educative.io/courses/data-structures-coding-interviews-java/g75kWBBLxPG#2.1-Max-Heap-Property:)

This property states that all the parent node keys must be greater than or equal to their child node keys. So the root node, in this case, will always contain the largest element present in the Heap. If Node A has a child node B, then,



#### 2.2 Min Heap Property:[**#**](https://www.educative.io/courses/data-structures-coding-interviews-java/g75kWBBLxPG#2.2-Min-Heap-Property:)

In Min Heap, all the parent node keys are less than or equal to their child node keys. This goes without saying that the rule will apply to all children of the node. So the root node, in this case, will always contain the smallest element present in the Heap. If Node A has a child node B, then,



In the next lesson, we will learn about practical applications of heaps.

# Why Use Heaps?

The lesson highlights the significant applications and reasons for picking Heaps Data Structure while programming.

**We'll cover the following**

* + [Where are Heaps Used?](https://www.educative.io/courses/data-structures-coding-interviews-java/xV6JB5V7n0l#Where-are-Heaps-Used?)

## Where are Heaps Used?[#](https://www.educative.io/courses/data-structures-coding-interviews-java/xV6JB5V7n0l#Where-are-Heaps-Used?)

Just like other data structures, Heaps are also used in many computing algorithms. The major uses of Heaps are elaborated below:

1. **Order statistics:** Heaps are primarily used for efficiently finding the smallest or largest element in an array.
2. **Priority Queues:** Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete(), extractmax(), and decreaseKey() operations in O(logn)*O*(*logn*) time. Binomoial Heaps and Fibonacci Heaps are variations of Binary Heaps. These variations also perform union() in O(logn)*O*(*logn*) time, which is an O(n)*O*(*n*) operation in a Binary Heap. Heap-implemented priority queues are used in Graph algorithms like Prim’s Algorithm and Dijkstra’s algorithm.

Root Popped Out! 2 Priority Queue Over!

**8** of 8

1. **Sorting:** HeapSort uses the Heap data structure to sort values in exactly the same way as TreeSort used a Binary Search Tree.  
   Each insert() and delete() operation is O(logN). At the very worst - the heap does not always have all N values in it, so the complexity is certainly no greater than O(NlogN). This is better than the worst-case for TreeSort, which–because you might build a degenerate Binary Search Tree-- is O(N\*N).  
   HeapSort is especially useful for sorting arrays because Heaps, unlike almost all other types of trees - are usually implemented in arrays, not as linked data structures!

In the next lesson, we will learn how to represent the binary tree heap structure using an array!

**Heap Representation in Arrays**

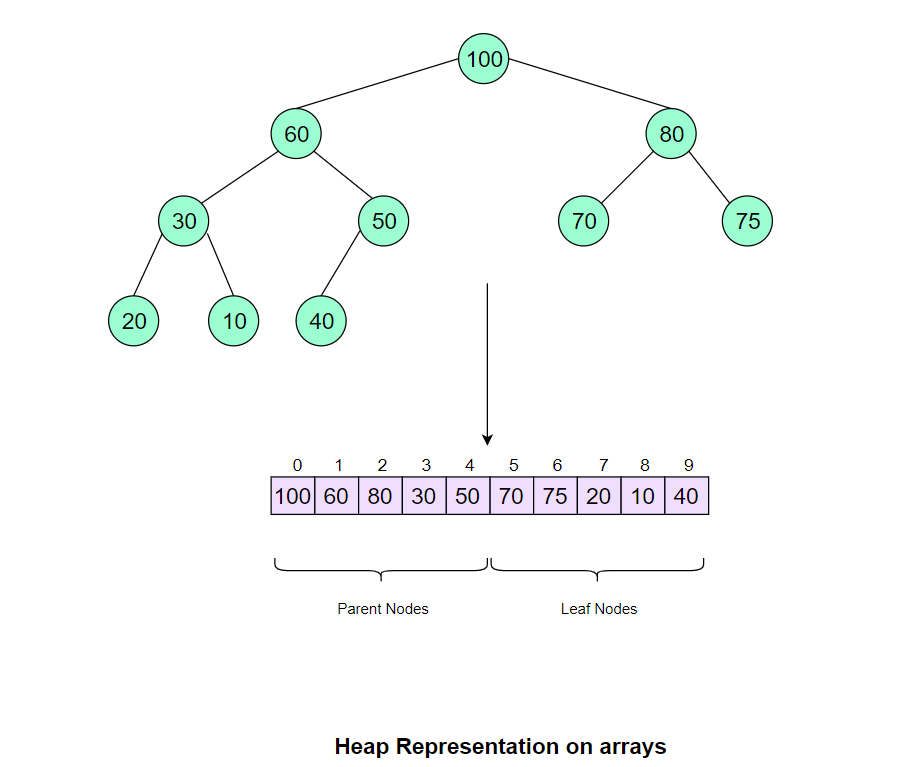
The lesson elaborates how to Use Arrays for the Implementation of Heaps.

Heaps can be implemented using Arrays. The contents of a heap with n nodes are stored in such a way that all the parent nodes occur in the first half of array (n/2), while the leaves are present at the last n/2 positions. So the last parent will be at the n/2th position.

The node at the *kth* position will have its children placed as follows:

* The Left child at **2k+1**
* The Right child at **2k+2**.

See the figure below to see how nodes are mapped on the array:



In the figure above, you can see that all of the parent nodes are present in the first half of the array, with the last parent at the n/2th position (i.e. 4th index), whereas the children nodes appear on the second half. The following properties also hold:

Left Child = 2k+1*LeftChild*=2*k*+1

Right Child = 2k+2*RightChild*=2*k*+2

In the next chapter, we will take a detailed look at Max Heap and see how it is different from Min Heap.

# Max Heap: Introduction

This lesson will give a brief introduction about Max Heap and how elements are inserted or removed from Max-Heap.

**We'll cover the following**

* + [Building a Max-Heap](https://www.educative.io/courses/data-structures-coding-interviews-java/B81K0P3EEGN#Building-a-Max-Heap)
  + [Implementing a Max-Heap](https://www.educative.io/courses/data-structures-coding-interviews-java/B81K0P3EEGN#Implementing-a-Max-Heap)
  + [Insertion in Max-Heap](https://www.educative.io/courses/data-structures-coding-interviews-java/B81K0P3EEGN#Insertion-in-Max-Heap)
  + [Removing an Element from a Max-Heap](https://www.educative.io/courses/data-structures-coding-interviews-java/B81K0P3EEGN#Removing-an-Element-from-a-Max-Heap)

## Building a Max-Heap[#](https://www.educative.io/courses/data-structures-coding-interviews-java/B81K0P3EEGN#Building-a-Max-Heap)

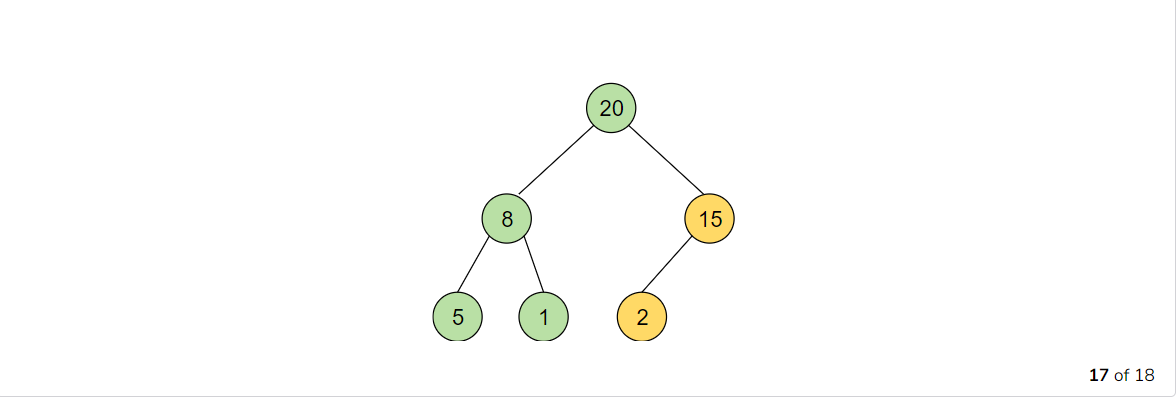
As mentioned in the previous lesson, a Max Heap follows the Max Heap property, which means the key at the parent node is always greater than keys at both child nodes. The following steps illustrate how we build a Max Heap:

1. Create a new node at the end of the heap.
2. Assign a new value to the node.
3. Compare the value of this child node with its parent.
4. If the value of the parent is less than that of the child, then swap them.
5. Repeat steps 3 & 4 until the Heap property holds.

## Implementing a Max-Heap[#](https://www.educative.io/courses/data-structures-coding-interviews-java/B81K0P3EEGN#Implementing-a-Max-Heap)

Heaps can be implemented using arrays. Initially, elements are placed in nodes in the same order as they appear in the array. Then a function is called over the whole Heap in a bottom-up manner, which “Max Heapifies” this Heap so that the Heap property is satisfied on all nodes.

When we say bottom-up we mean the function starts from the last parent node present at the n/2th position of the array, and it checks if the values at the child nodes are greater than the parent nodes. See how we constructed a Heap in the following illustration:

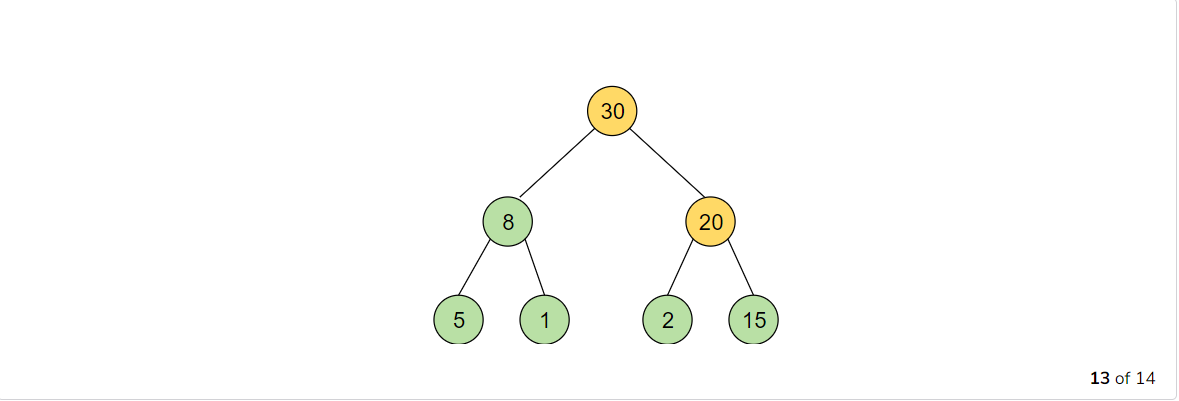
**17** of 18

## Insertion in Max-Heap[#](https://www.educative.io/courses/data-structures-coding-interviews-java/B81K0P3EEGN#Insertion-in-Max-Heap)

If you want to insert a new element in the Max Heap, you will have to follow a list of steps to make sure the Heap property still holds after adding the element. Here’s the list of steps that you will perform:

1. Create a new child node at the end of the heap.
2. Place the new key at that node.
3. Compare the value with its parent node key.
4. If the key is greater than the key at the parent node, swap values.
5. If both keys at the children nodes are greater than the parent node key, pick the larger one and see if the Heap property is satisfied.
6. Repeat until you reach the root node.

For better understanding, here’s the visual representation of what we just said:

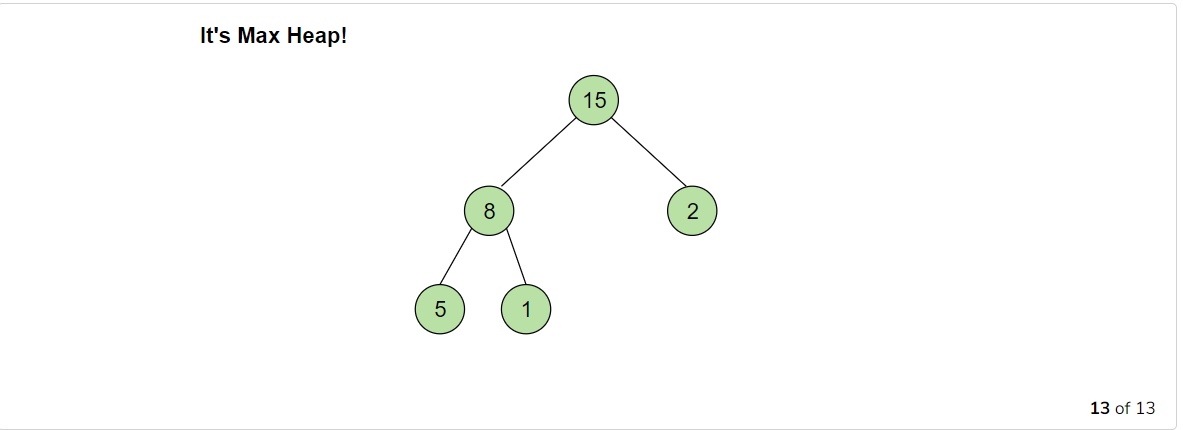
**13** of 14

## Removing an Element from a Max-Heap[#](https://www.educative.io/courses/data-structures-coding-interviews-java/B81K0P3EEGN#Removing-an-Element-from-a-Max-Heap)

Deletion in a Max-Heap is mainly performed when you want to remove the largest element. In most of the cases, the purpose of a Heap is to work as a priority queue. As an example here, we will take the case of deleting the biggest element here as we are discussing Max Heaps. Given below is the list of steps you will follow to make sure the Heap Property still holds after deleting the root element:

1. Delete the root node
2. Move the key of the last child node at the last level to the root
3. Now compare the key with its children
4. If the key is smaller than the key at any of the child nodes, swap values
5. If both keys at the children nodes are greater than the parent node key, pick the larger one and see if the heap property is satisfied
6. Repeat until you reach the last level

For better understanding, here’s the visual representation of what we just said:

It's Max Heap!

**13** of 13

In the next chapter, we will implement Max Heap and add all these scenarios that we have discussed above in the code.

# Max Heap (Implementation)

How is Max-Heap implemented in Java? Let's find out in this lesson.

**We'll cover the following**

* + [Implementation](https://www.educative.io/courses/data-structures-coding-interviews-java/JPX181R2Vzv#Implementation-)
    - [Explanation](https://www.educative.io/courses/data-structures-coding-interviews-java/JPX181R2Vzv#Explanation-)
  + [If this Code had a Face](https://www.educative.io/courses/data-structures-coding-interviews-java/JPX181R2Vzv#If-this-Code-had-a-Face-)
  + [Time Complexity](https://www.educative.io/courses/data-structures-coding-interviews-java/JPX181R2Vzv#Time-Complexity)

## Implementation [#](https://www.educative.io/courses/data-structures-coding-interviews-java/JPX181R2Vzv#Implementation-)

Now that we have discussed the important Max Heap functions, let’s move on to implementing them in Java.

import java.util.Arrays;

class Heap {

private void maxHeapify(int[] heapArray, int index, int heapSize){

int largest = index;

while (largest < heapSize / 2){ // check parent nodes only

int left = (2 \* index) + 1; //left child

int right = (2 \* index) + 2; //right child

if (left < heapSize && heapArray[left] > heapArray[index]){

largest = left;

}

if (right < heapSize && heapArray[right] > heapArray[largest]){

largest = right;

}

if (largest != index){ // swap parent with largest child

int temp = heapArray[index];

heapArray[index] = heapArray[largest];

heapArray[largest] = temp;

index = largest;

}

else

break; // if heap property is satisfied

} //end of while

}

public void buildMaxHeap(int[] heapArray, int heapSize)

{

// swap largest child to parent node

for (int i = (heapSize - 1) / 2; i >= 0; i--){

maxHeapify(heapArray, i, heapSize);

}

}

public static void main(String[] args) {

int[] heapArray = { 1, 4, 7, 12, 15, 14, 9, 2, 3, 16 };

System.out.println("Before heapify: "+Arrays.toString(heapArray));

new Heap().buildMaxHeap(heapArray, heapArray.length);

System.out.println("After heapify: "+Arrays.toString(heapArray));

}

}

### Explanation [#](https://www.educative.io/courses/data-structures-coding-interviews-java/JPX181R2Vzv#Explanation-)

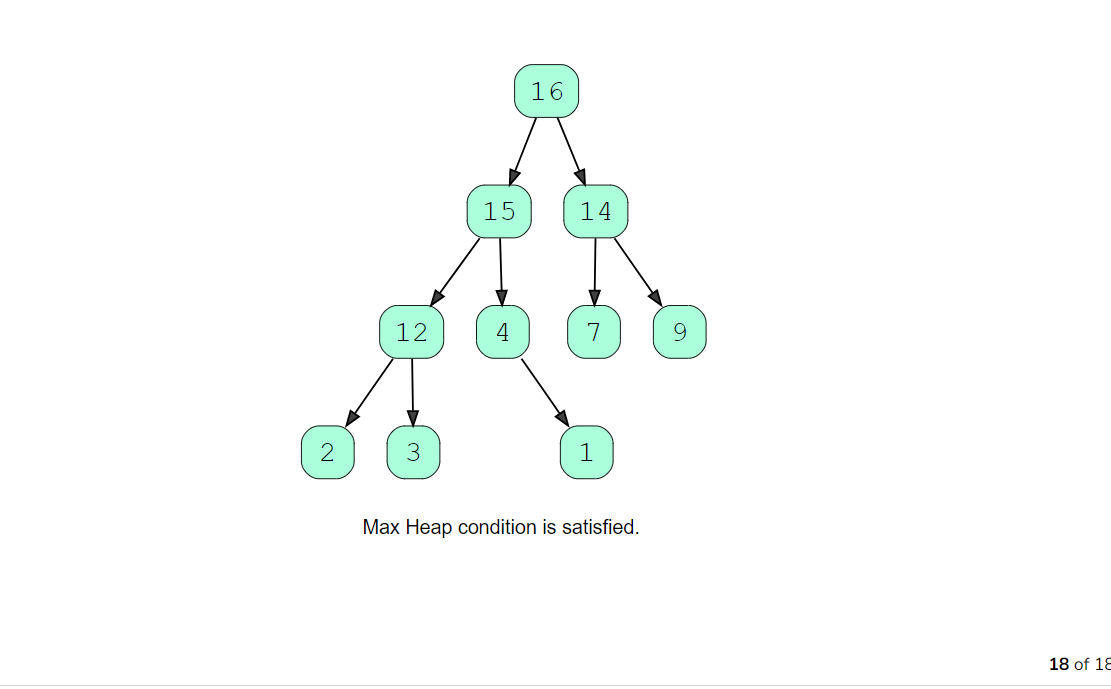
This code covers all the cases that we discussed in the previous chapter. Let’s look at each function one by one and see what’s going on:

* **BuildHeap():** It takes the array and starts from the last child node at the last level, then passes it to MaxHeapify for comparison.
* **MaxHeapify():** This function takes the node value and compares it with the key at the parent node, and swaps them if the condition below stands true. The while loop will take care of parent/child comparison during different level swapping till the max heap rule holds good.

Child Node >= Parent Node

The while loop makes sure that the nodes keep swapping until the Heap property is satisfied, so we basically call MaxHeapify(); at each small level to achieve **Max Heap.**

## If this Code had a Face [#](https://www.educative.io/courses/data-structures-coding-interviews-java/JPX181R2Vzv#If-this-Code-had-a-Face-)

Max Heap condition is satisfied.

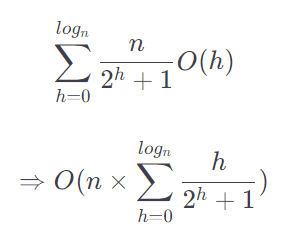
**18** of 18

## Time Complexity[#](https://www.educative.io/courses/data-structures-coding-interviews-java/JPX181R2Vzv#Time-Complexity)

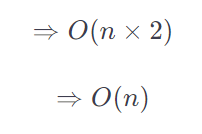
The worst-case time complexity of maxHeapify() is O(lgn) because we start with the rightmost parent node in the heap, then move left and then up until we reach the root node.

In buildMaxHeap(), the heapify function is called O(n) times. Therefore, the overall time complexity of building a **Heap** seems to be O(n(lgn))), but this is a **very loose upper bound**. A more accurate and **tight upper-bound** for the build heap operation is O(n). Let me explain how we can calculate this.

The heapify function has different time complexity at each level of the tree i.e it will be O(1) at the leaf node and O(lg(n))) at the root. So, the worst-case time complexity of heapify at each node is O(h) where h is the height of the node in the heap. The number of nodes for any binary given tree with height h is given by n/{2^h + 1}​. According to these measures, the total complexity can be calculated by the following expression:



When the above summation approaches ∞, it converges to 2. The expression thus becomes:



Hence, the time complexity of building a heap is O(n)

You see, just with the help of two simple functions, we have implemented a whole data structure.

And now that we have covered Max Heap, implementing a Min Heap will not be a problem. So, that’s what we are going to study in the next lesson. See you!

# Min Heap: An Introduction

This lesson will give a brief introduction about Min Heap, and how elements are inserted or removed from them.

**We'll cover the following**

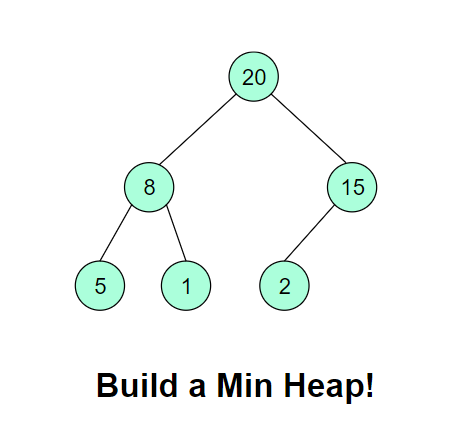
* + [Building a Min-Heap](https://www.educative.io/courses/data-structures-coding-interviews-java/NE0MrPKAnWN#Building-a-Min-Heap)
  + [Insertion in Min Heap](https://www.educative.io/courses/data-structures-coding-interviews-java/NE0MrPKAnWN#Insertion-in-Min-Heap)
  + [Deletion in Min Heap](https://www.educative.io/courses/data-structures-coding-interviews-java/NE0MrPKAnWN#Deletion-in-Min-Heap)

## Building a Min-Heap[#](https://www.educative.io/courses/data-structures-coding-interviews-java/NE0MrPKAnWN#Building-a-Min-Heap)

As mentioned in the previous lesson, a Min Heap follows the Min Heap property, which means the key at the parent node is always smaller than keys at both children nodes.

Heaps can be implemented using arrays. Initially, elements are placed in nodes in the same order as they appear in the array. Then a function is called over the whole Heap in a bottom-up manner, which “Min Heapifies” this Heap so that the Heap property is satisfied on all nodes.

When we say bottom-up, we mean the function starts from the last parent node present in the n/2th position of the array, and checks if the values at the children nodes are smaller than the parent node. If yes, then swap the values; if no, then move to the next parent node. See how we constructed a Heap in the following illustration:

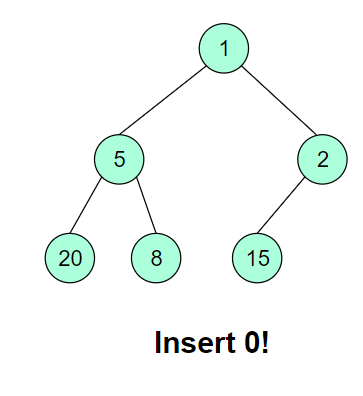


**Insertion in Min Heap**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/NE0MrPKAnWN#Insertion-in-Min-Heap)

If you want to insert a new element in a Min Heap, you will have to follow a list of steps to make sure the Heap property still holds after adding the element. Here’s the list of steps that you will perform:

1. Create a new child node at the end of the heap
2. Place the new key at that node
3. Compare the value with its parent node key
4. If the key is smaller than the key at the parent node, swap values
5. If both keys at the children nodes are smaller than the parent node key, then pick the smallest one and see if the Heap property is satisfied.
6. Repeat until you reach the root node

For better understanding, here’s the visual representation of what we just said:

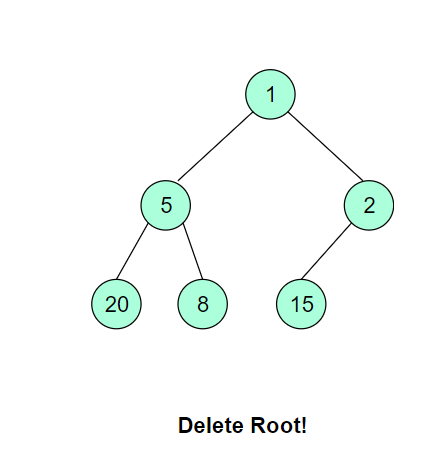


**Deletion in Min Heap**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/NE0MrPKAnWN#Deletion-in-Min-Heap)

Deletion is performed in the same way as in Max Heap. We will take the case of deleting the smallest element here as we are discussing Min Heaps. Given below is the list of steps you will follow to make sure the Heap property still holds after deleting the root element:

1. Delete the root node
2. Move the key of the last child node (at the last level) to the root
3. Now compare the key with its children
4. If the key is greater than the key at any of the children nodes, swap values
5. If both keys at children nodes are smaller than the parent node key, pick the smallest one and see if the Heap property is satisfied.
6. Repeat until you reach the last level

For better understanding, here’s the visual representation of what we just said above:



Since we have already implemented Max Heap in the previous lesson, implementing Min Heap should be a piece of cake! Let’s look at its implementation in the next lesson.

# Min Heap (Implementation)

How is a Min Heap implemented in Java? Let's find out in this lesson.

**We'll cover the following**

* + [Implementation](https://www.educative.io/courses/data-structures-coding-interviews-java/RMw7Vg8OyQO#Implementation)
    - [Explanation:](https://www.educative.io/courses/data-structures-coding-interviews-java/RMw7Vg8OyQO#Explanation:)
  + [If this Code had a Face](https://www.educative.io/courses/data-structures-coding-interviews-java/RMw7Vg8OyQO#If-this-Code-had-a-Face)
  + [Time Complexity of Min-Heap](https://www.educative.io/courses/data-structures-coding-interviews-java/RMw7Vg8OyQO#Time-Complexity-of-Min-Heap)

## Implementation[#](https://www.educative.io/courses/data-structures-coding-interviews-java/RMw7Vg8OyQO#Implementation)

Now that we have discussed all the scenarios of a Min Heap, let’s move forward and implement these scenarios in the following code. Run and test the code on multiple outputs to see if it returns the elements in correct order every time? Try it!

import java.util.Arrays;

class Heap {

private void minHeapify(int[] heapArray, int index, int heapSize) {

int smallest = index;

while (smallest < heapSize / 2) { // check parent nodes only

int left = (2 \* index) + 1; //left child

int right = (2 \* index) + 2; //right child

if (left < heapSize && heapArray[left] < heapArray[index]) {

smallest = left;

}

if (right < heapSize && heapArray[right] < heapArray[smallest]) {

smallest = right;

}

if (smallest != index) { // swap parent with smallest child

int temp = heapArray[index];

heapArray[index] = heapArray[smallest];

heapArray[smallest] = temp;

index = smallest;

} else {

break; // if heap property is satisfied

}

} //end of while

}

public void buildMinHeap(int[] heapArray, int heapSize) {

// swap smallest child to parent node

for (int i = (heapSize - 1) / 2; i >= 0; i--) {

minHeapify(heapArray, i, heapSize);

}

}

public static void main(String[] args) {

int[] heapArray = { 31, 11, 7, 12, 15, 14, 9, 2, 3, 16 };

System.out.println("Before heapify: "+Arrays.toString(heapArray));

new Heap().buildMinHeap(heapArray, heapArray.length);

System.out.println("After heapify: "+Arrays.toString(heapArray));

}

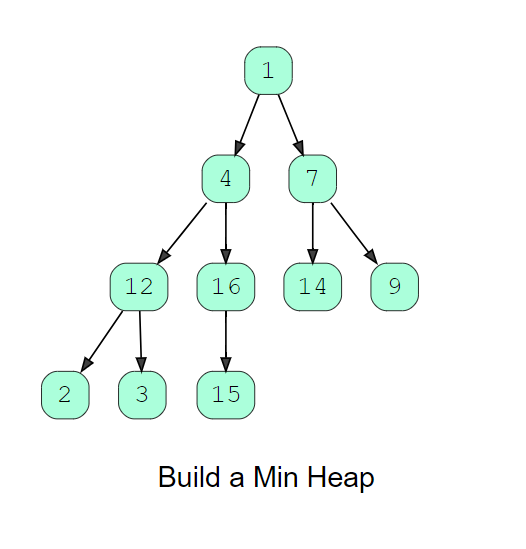
}

**Explanation:**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/RMw7Vg8OyQO#Explanation:)

This code covers all the cases that we discussed in the previous chapter. Let’s look at each function one by one and see what’s going on:

* **BuildHeap():** It takes the array and starts from the last parent node (at the second last level), then passes it to MinHeapify for comparison.
* **MinHeapify():** This function takes the node index and compares the key (the parent node) with its child nodes, and swaps them if the child node < the parent node. The while loop makes sure that the nodes keep swapping until we reach the last index and Heap property is satisfied throughout the Heap!

**If this Code had a Face**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/RMw7Vg8OyQO#If-this-Code-had-a-Face)



## Time Complexity of Min-Heap[#](https://www.educative.io/courses/data-structures-coding-interviews-java/RMw7Vg8OyQO#Time-Complexity-of-Min-Heap)

The overall time complexity of building the Heap in a Min Heap is the same in as a Max Heap: O(n). You can refer to [this lesson](https://www.educative.io/collection/page/5642554087309312/5724822843686912/5161164958859264) for the complete calculation.

You see, just like Max Heap, we have implemented a whole data structure with the help of only two major basic functions. And now that we have covered both the implementations, let’s try to solve some practice questions using Heaps in the next lessons!