Introduction to Two Heaps

Let’s go over the Two Heaps pattern, its real-world applications, and some problems we can solve with it.

We'll cover the following...

About the pattern

The **two heaps** pattern is a versatile and efficient approach used to solve problems involving dynamic data processing, optimization, and real-time analysis. As the name suggests, this pattern maintains two heaps, which could be either two min heaps, two max heaps, or a min heap and a max heap. Exploiting the heap property, the two heaps pattern is a preferred technique for various problems to implement computationally efficient solutions. For a heap containing n elements, inserting or removing an element takes $O(logn)$ time, while accessing the element at the root is done in $O(1)$ time. The root stores the smallest element in the case of a min heap and the largest element in the case of a max heap.

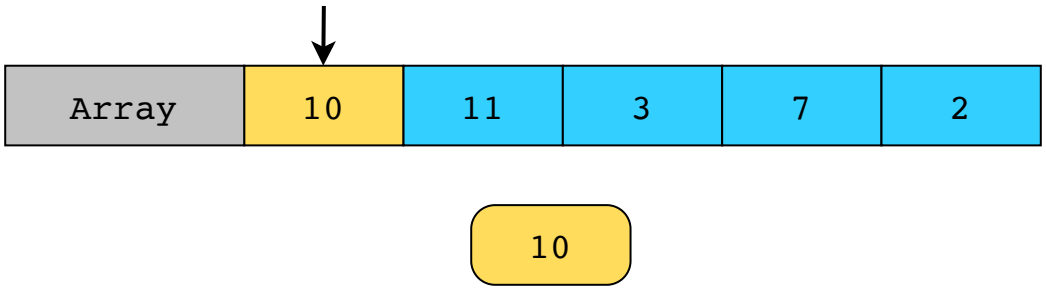
Let’s explore a few example scenarios to gain a better understanding. In some problems, we’re given a dataset and tasked to divide it into two parts to find the smallest value from one part and the largest value from the other part. To achieve this, we can build two heaps: one min heap and one max heap, from these two subsets of data. The root of the min heap will give us the smallest value from its corresponding dataset, while the root of the max heap will provide the largest value from its dataset. In

cases where we need to find the two largest numbers from two different datasets, we'll use two max heaps to store and manage these datasets. Similarly, to find the two smallest numbers from two different datasets, we would use two min heaps. These examples illustrate the versatility of using min heaps and max heaps to efficiently solve different types of problems by facilitating quick access to the smallest or largest values as required.

The following illustration demonstrates how we can build a min heap or a max heap, and how they can be used to solve several tasks, e.g., finding the smallest or largest element from some data:

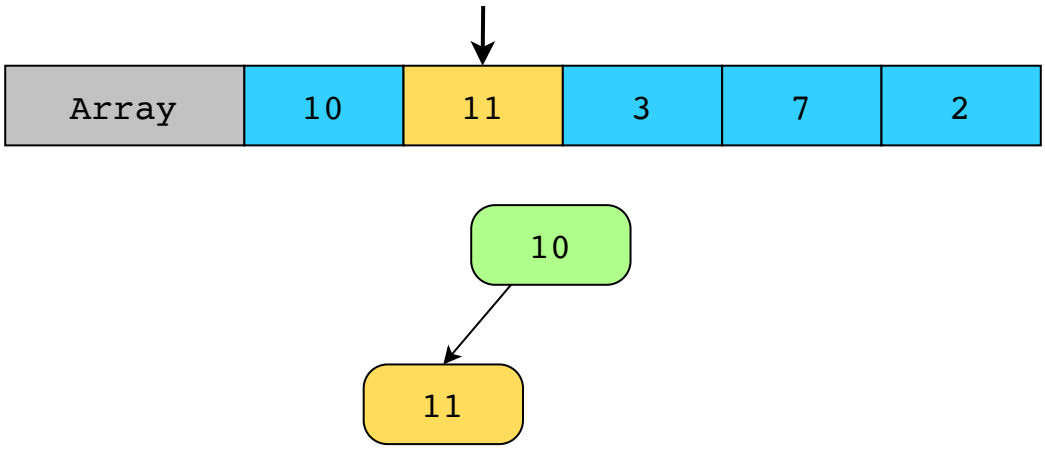
We will first construct a min-heap from the array below.

Node **10** is inserted into the heap as the root.

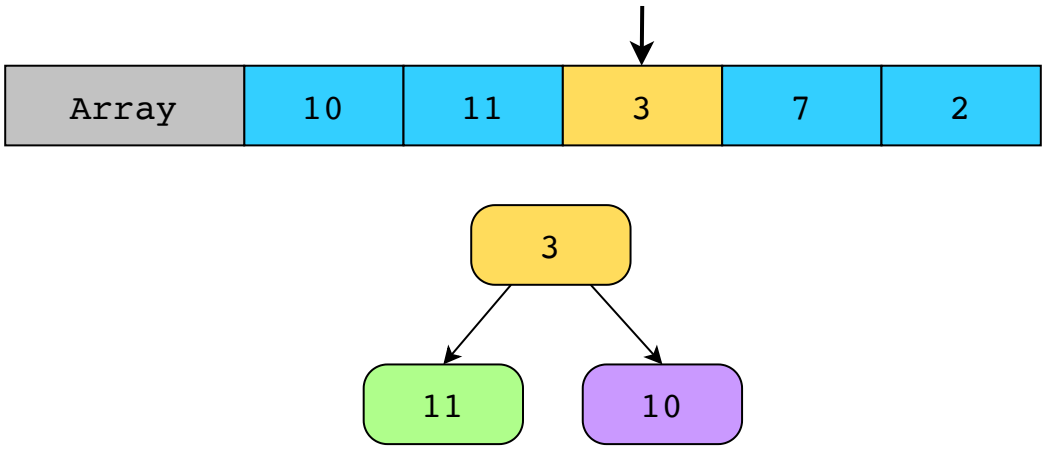


2 of 13

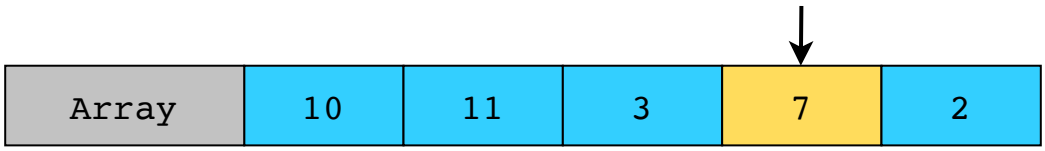
Since **11** > **10**, and as the smallest element i.e., **10** is the root of the min-heap, **11** is set up as the left child of the root node.

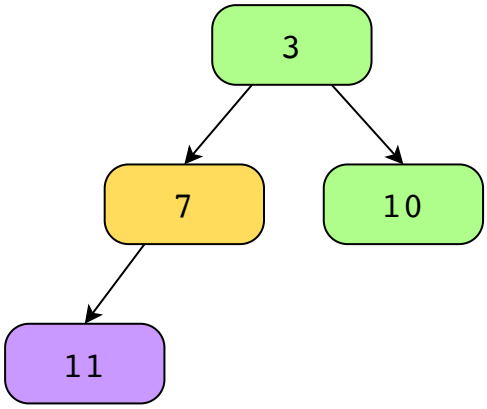


We insert **3**, and since **3** < **10**, the positions of node **3** and node **10** are swapped to make node **3** the new root.



Since **3** < **7** < **11**, the new node **7** is placed as left child of **3** and **11** is made the left child of **7**.

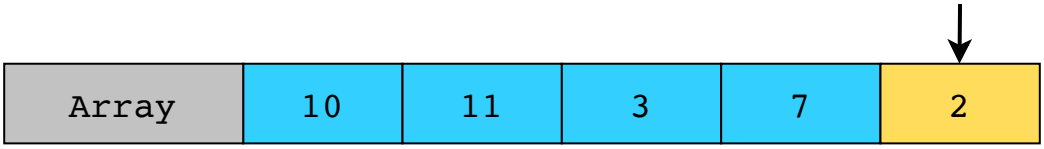


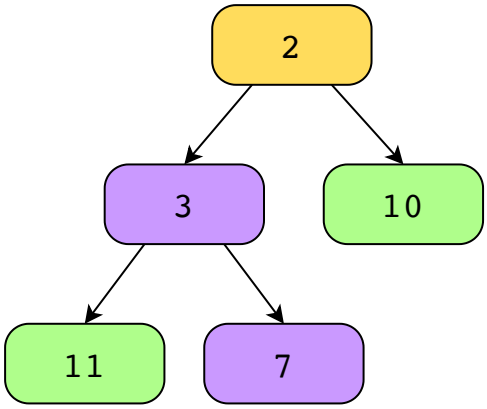


5 of 13

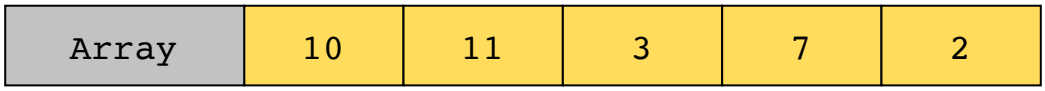
Since **2** is less than the root element **3**, **2** will reside as the root of this min-heap. The nodes **3** and **7** are adjusted accordingly.

The min-heap representation of the array is now complete.

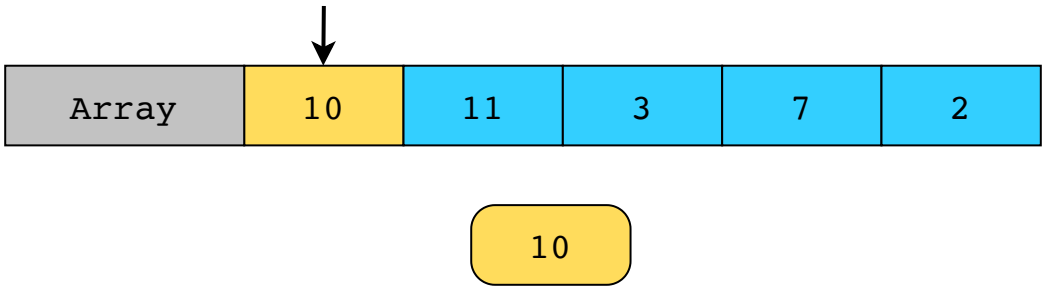




We will now construct a max-heap from the same array.

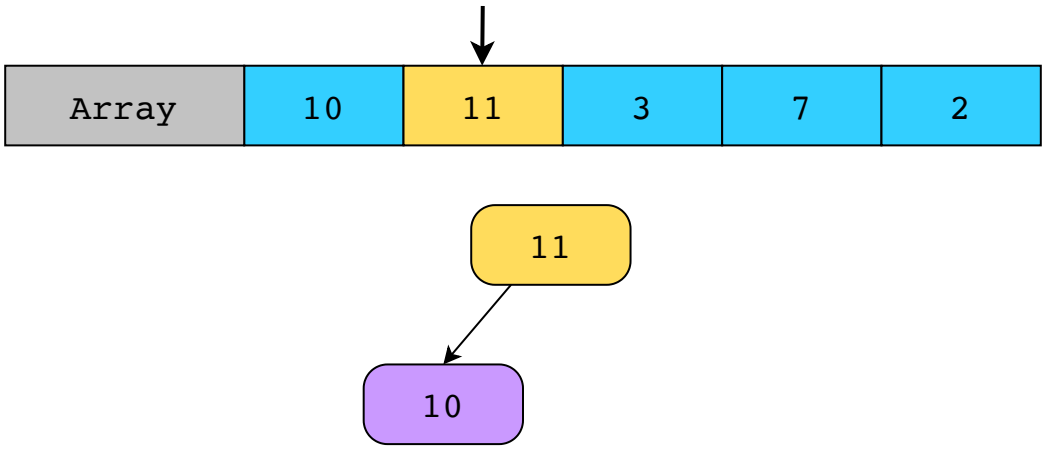


Node **10** is inserted into the heap as the root.

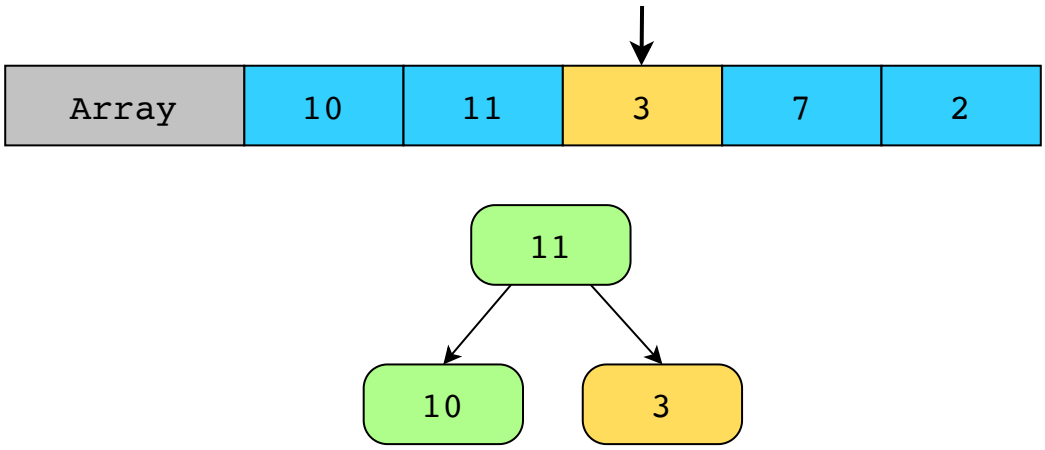


8 of 13

Since **11** > **10**, **11** is set as the root of the max-heap and **10** is made its left child.

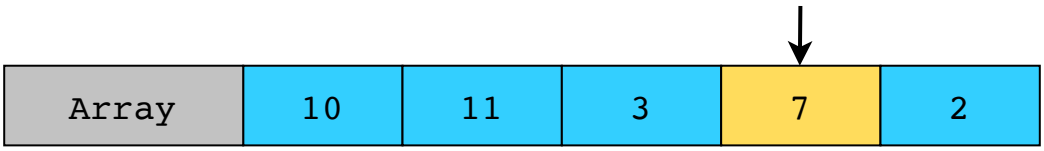


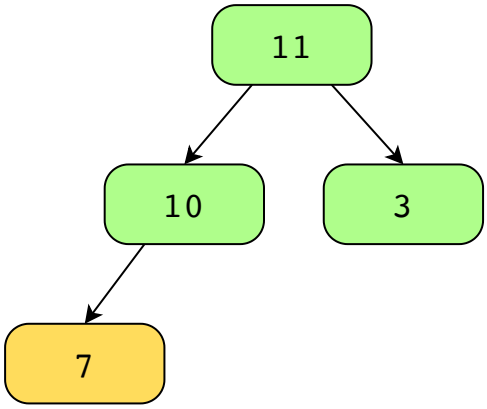
Since **3** < **11**, node **3** is inserted as the left child of node **11**.



10 of 13

Since **7** < **10**, node **7** is inserted as the left child of node **10**.

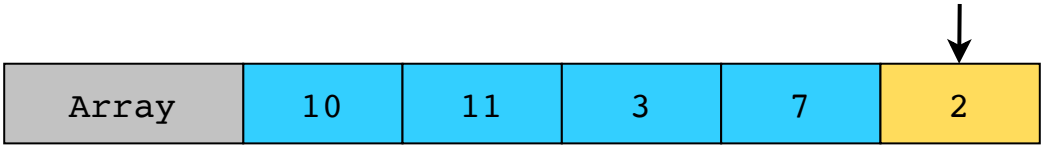


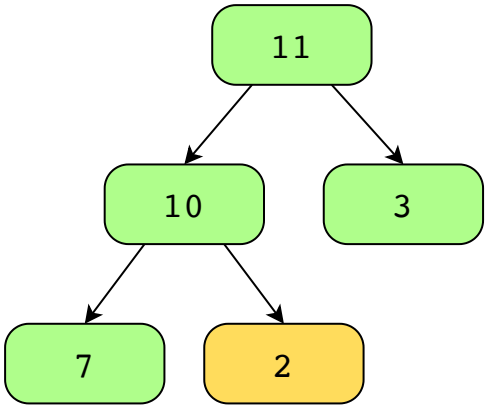


11 of 13

Since **2** < **10**, node **2** is inserted as the right child of node **10**.

The max-heap representation of the array is now complete.





Here's the side-by-side view of both the min-heap and max-heap representations of the array.

Note that the top of **min-heap** represents the smallest value in the data, whereas the top of **max-heap** represents the largest value in the data.