Heap Sort

[1, 7, 2, 9, 3, 10, 8, 6]

[_, _, _, _, _, _, _, _]

Fill each spot one by one from right to left

The idea: Fill each spot one by one from right to left

Heap Sort

[1, 7, 2, 9, 3, ~~10~~, 8, 6]

[_, _, _, _, _, _, _,10]

Fill each spot one by one from right to left

Heap Sort

[1, 7, 2, ~~9~~, 3, ~~10~~, 8, 6]

[_, _, _, _, _, _, 9,10]

Fill each spot one by one from right to left

Heap Sort

[1, 7, 2, ~~9~~, 3, ~~10~~, ~~8~~, 6]

[_, _, _, _, _, 8, 9,10]

Fill each spot one by one from right to left

Heap Sort

[1, ~~7~~, 2, ~~9~~, 3, ~~10~~, ~~8~~, 6]

[_, _, _, _, 7, 8, 9,10]

Fill each spot one by
one from right to left

Heap Sort

[1, ~~7~~, 2, ~~9~~, 3, ~~10~~, ~~8~~, ~~6~~]

[_, _, _, 6, 7, 8, 9,10]

Fill each spot one by
one from right to left

Heap Sort

[1, ~~7~~, 2, ~~9~~, ~~3~~, ~~10~~, ~~8~~, ~~6~~]

[_, _, 3, 6, 7, 8, 9,10]

Fill each spot one by one from right to left

Heap Sort

[1, ~~7~~, ~~2~~, ~~9~~, ~~3~~, ~~10~~, ~~8~~, ~~6~~]

[_, 2, 3, 6, 7, 8, 9,10]

Fill each spot one by one from right to left

Heap Sort

[~~1~~, ~~7~~, ~~2~~, ~~9~~, ~~3~~, ~~10~~, ~~8~~, ~~6~~]

[1, 2, 3, 6, 7, 8, 9,10]

Fill each spot one by one from right to left

Heap Sort

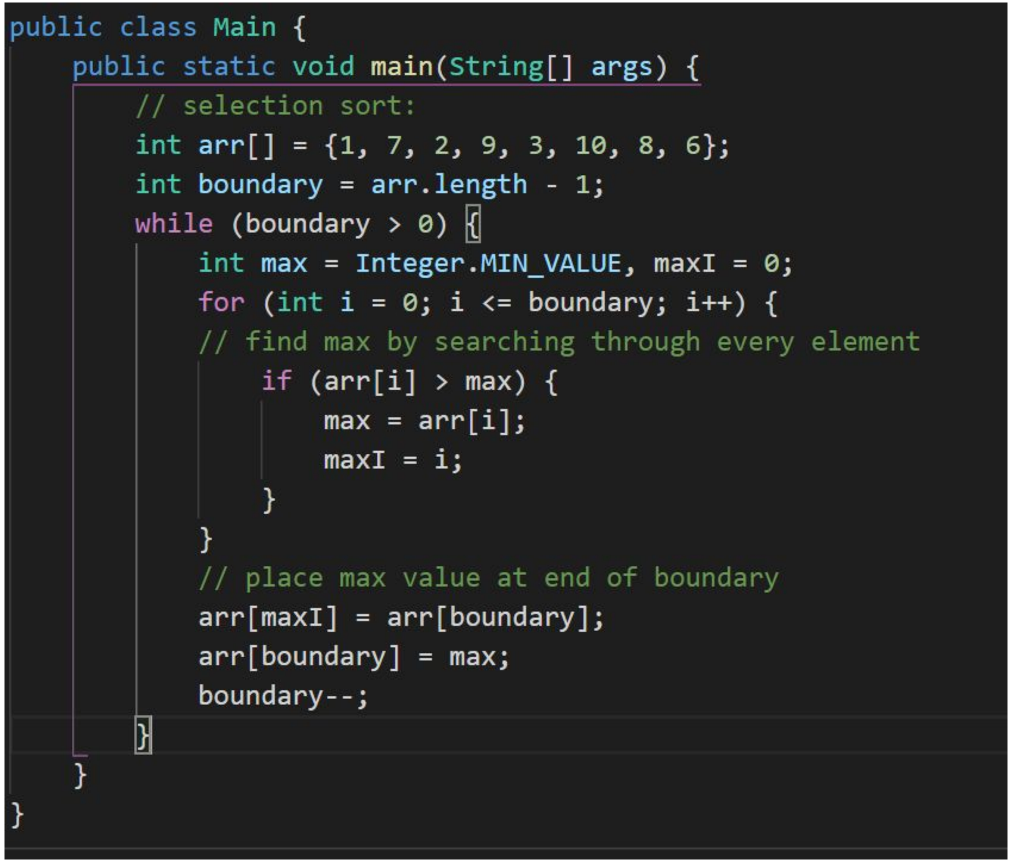
[~~1~~, ~~7~~, ~~2~~, ~~9~~, ~~3~~, ~~10~~, ~~8~~, ~~6~~]

[1, 2, 3, 6, 7, 8, 9,10]

Problem: How do we find the maximum each time efficiently?

Selection Sort: Too Slow ($O(n^2)$)

Selection sort finds max by searching through all elements



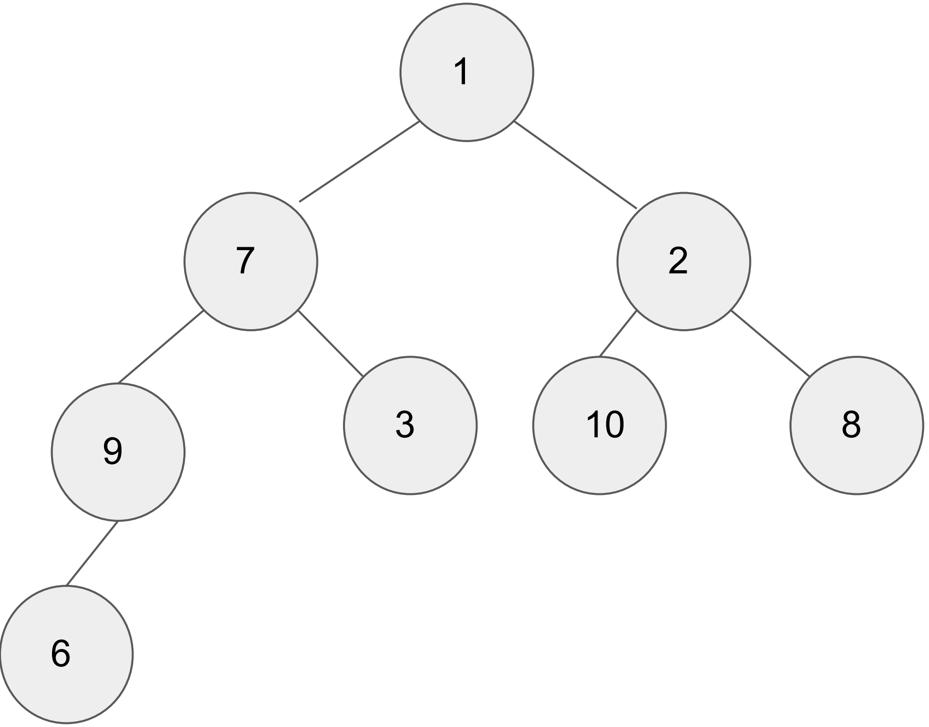
```java
public class Main {
    public static void main(String[] args) {
        // selection sort:
        int arr[] = {1, 7, 2, 9, 3, 10, 8, 6};
        int boundary = arr.length - 1;
        while (boundary > 0) {
            int max = Integer.MIN_VALUE, maxI = 0;
            for (int i = 0; i <= boundary; i++) {
            // find max by searching through every element
                if (arr[i] > max) {
                    max = arr[i];
                    maxI = i;
                }
            }
            // place max value at end of boundary
            arr[maxI] = arr[boundary];
            arr[boundary] = max;
            boundary--;
        }
    }
}
```

Max-Heap Data Structure!

We will use a data structure called a Max-Heap.

A Max-Heap is a full binary tree where every node is greater than or equal to both its children. Therefore, the root is the greatest value.

Construct a tree from array [1, 7, 2, 9, 3, 10, 8, 6]



The root is the variable at index 0 of the original array. The left child of each index *i* of the array is at 2*i* + 1 and the right child is at 2*i* + 2. The parent of each child *i* is at ⌊(*i* - 1) / 2⌋.

In simpler terms, the order is [root, left child, right child, left child of left child, right child of left child, left child of right child, right child of right child, left child of left child of left child...]

Heap Methods: parent, leftChild and rightChild

We now can turn the formulas in the previous slide into methods:

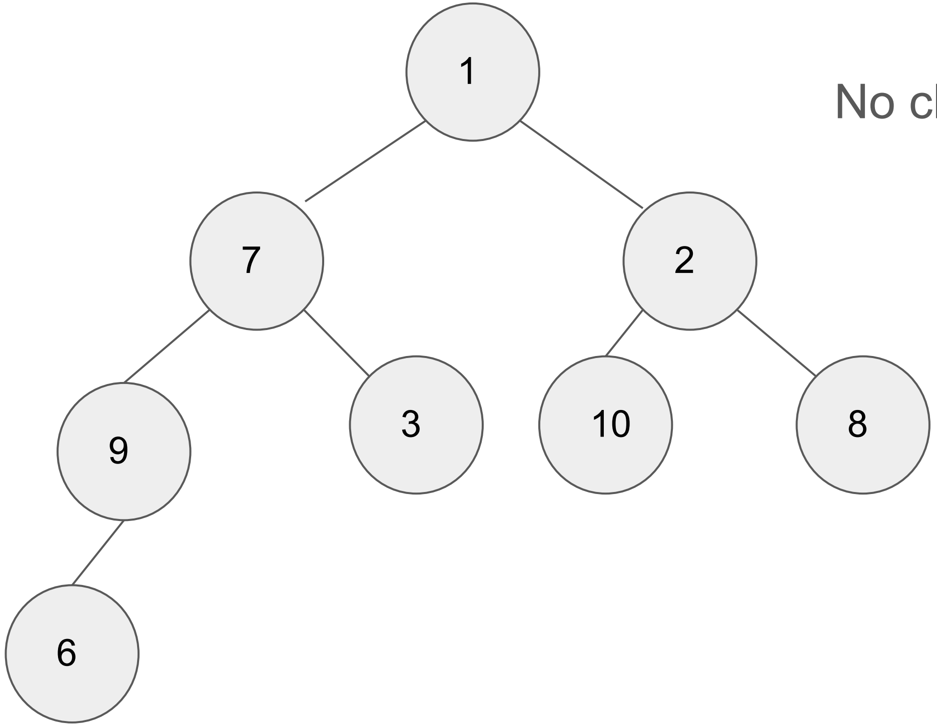
Building the Heap so it satisfies Max-Heap properties

We have to move down every node that has less value than one or both of its children. We find the node with the greatest value between the node, its left child, and its right child. If the max value is not the node, we swap it with the node, and continue the process. This method is called siftDown() (it is also sometimes called heapify()).

We will call this method for every non-leaf(which are all the elements from indices 0 to Math.floor((arr.length - 1) / 2) starting from the end to the root.

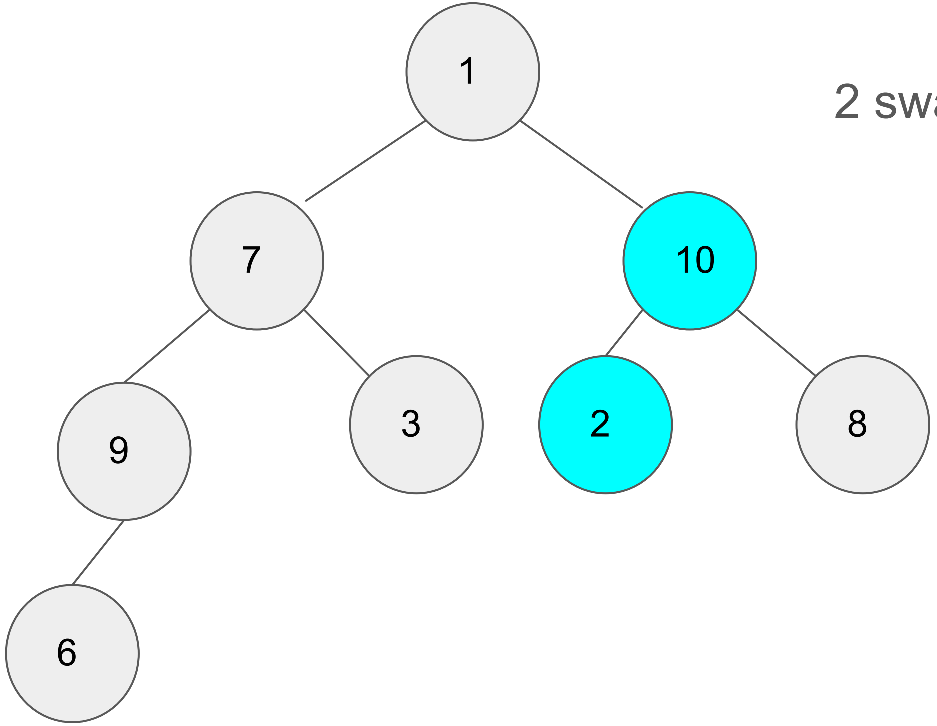
Let’s see this in action.

Sifting Down: 9



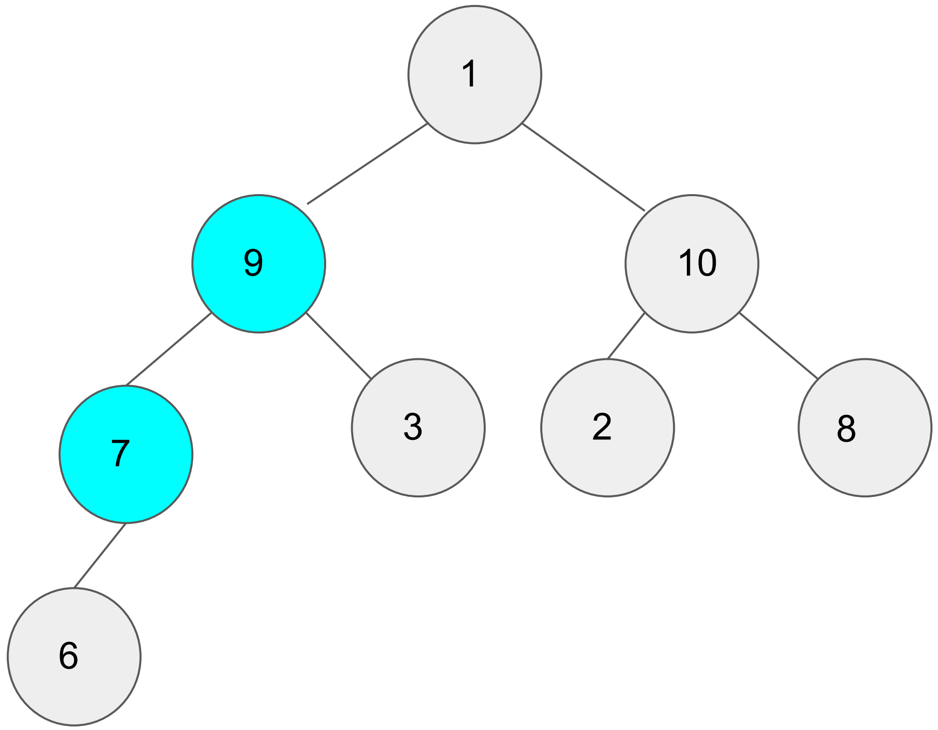
No change (9 is greater than 6, its child)

Sifting Down: 2



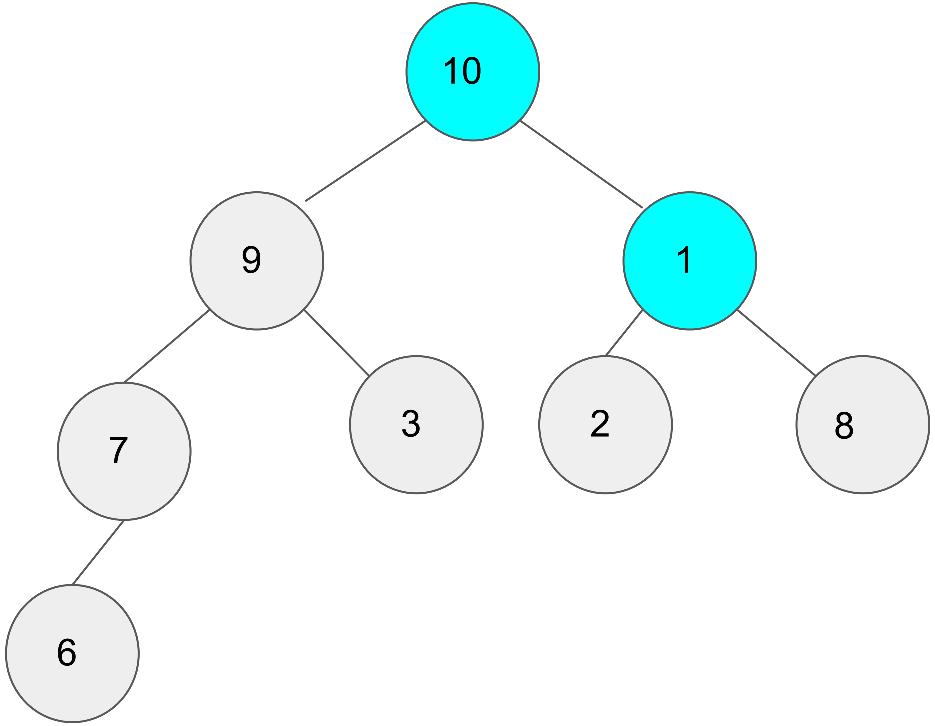
2 swaps with the greater of its children, 10.

Sifting Down: 7



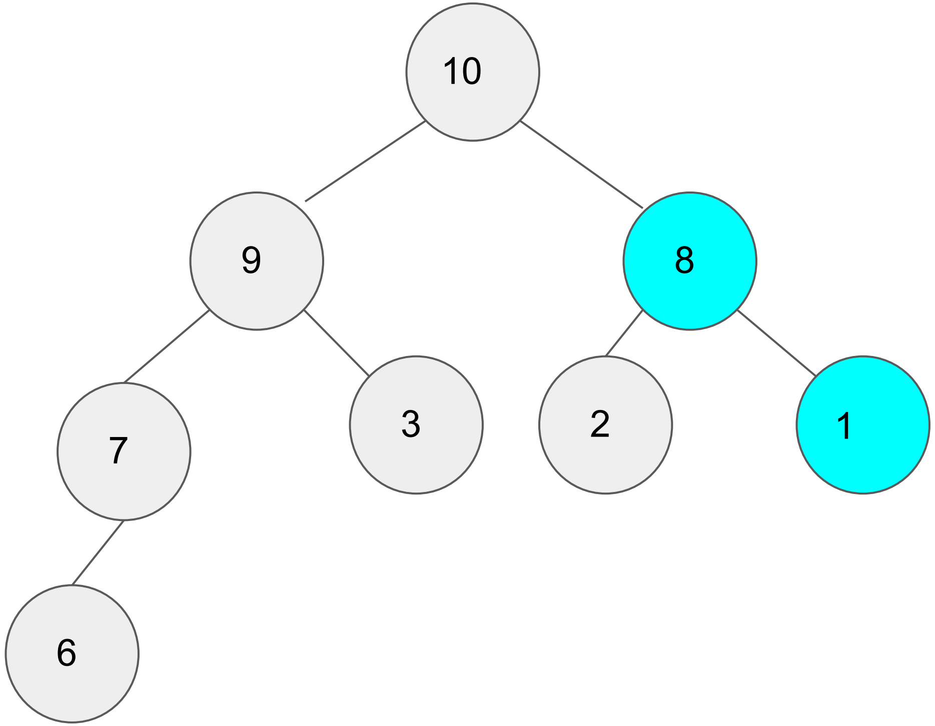
7 swaps with its greater child, 9.

Sifting Down: 1



1 swaps with its greater child, 10.

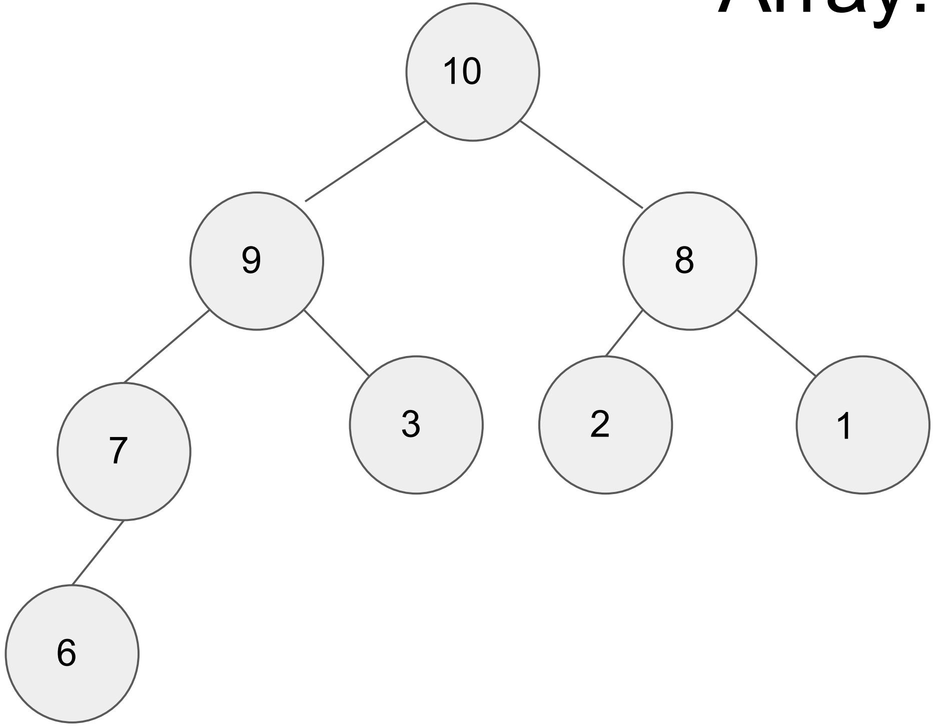
Sifting Down: 1



1 swaps with its greater child, 8.

Fully Built Max-Heap

Array: [10, 9, 8, 7, 3, 2, 1, 6]

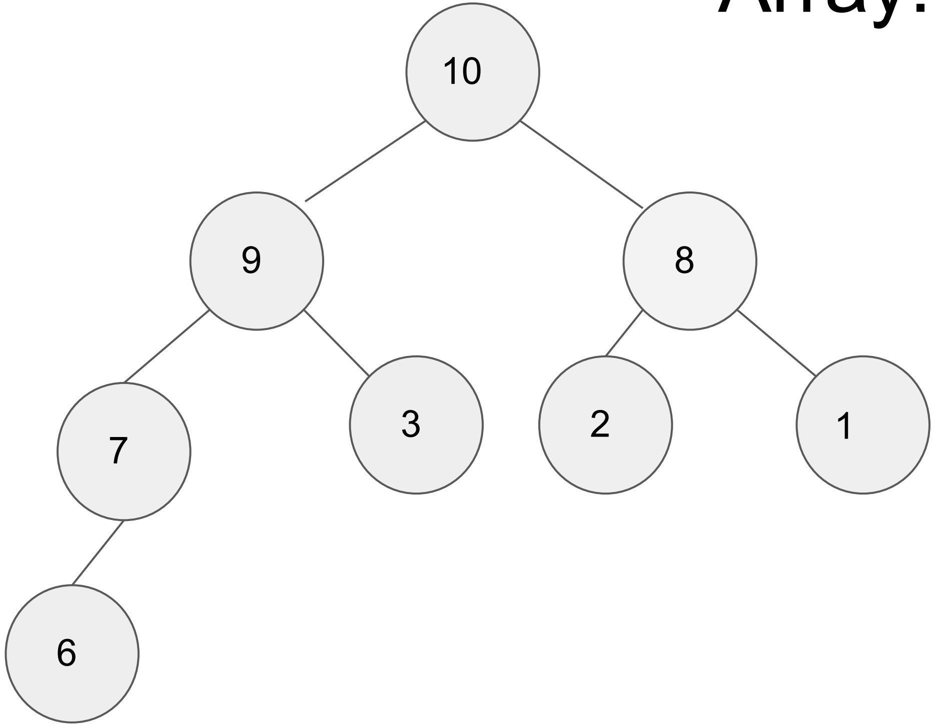


Extracting the Root (max value) and Sorting

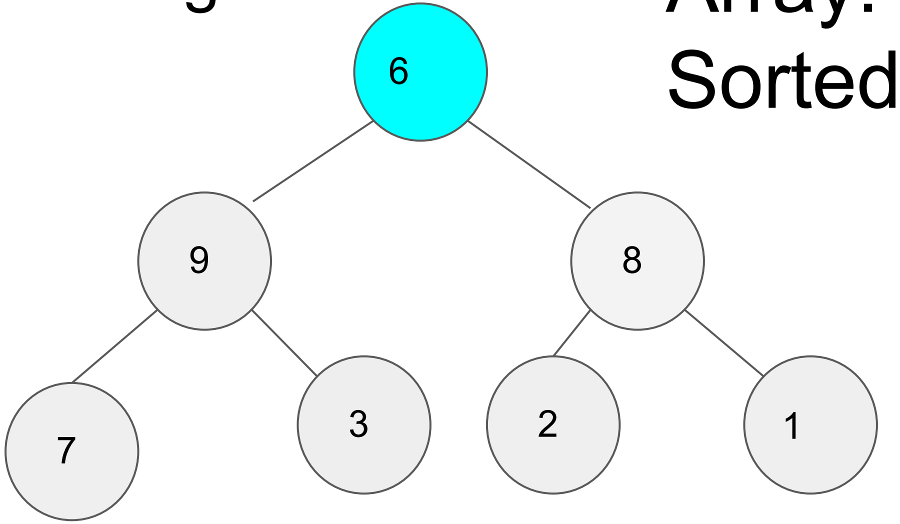
This method is pretty simple. We just swap the root with the last element in the array (moving the root into a boundary that will be the sorted region), and call siftDown(0). Then we decrease the size by 1. We repeat this while the size is > 0. By the end, we’ll get a sorted array! Let’s see how this works:

Our Fully Built Heap:

Array: [10, 9, 8, 7, 3, 2, 1, 6]



Replace the root with the last element and add root to sorted region

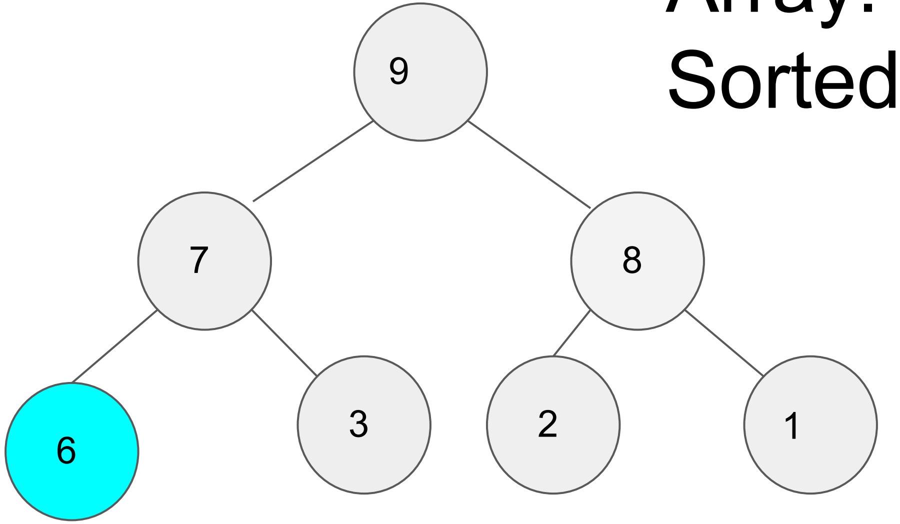


Array: [6, 9, 8, 7, 3, 2, 1]

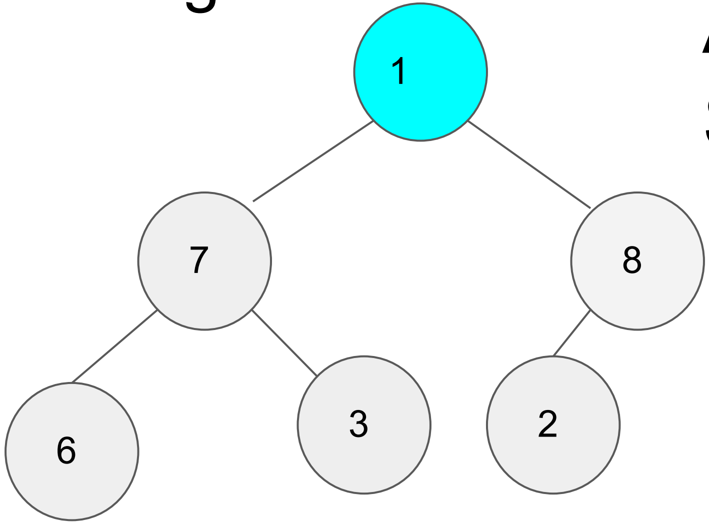
Sorted region: [10]

Sift Down the Root

Array: [9, 7, 8, 6, 3, 2, 1]
Sorted region: [10]



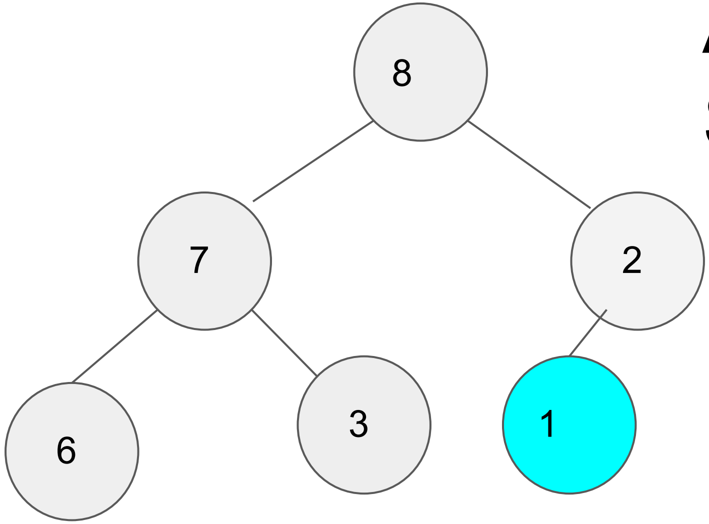
Replace the root with the last element and add root to sorted region



Array: [1, 7, 8, 6, 3, 2]

Sorted region: [9, 10]

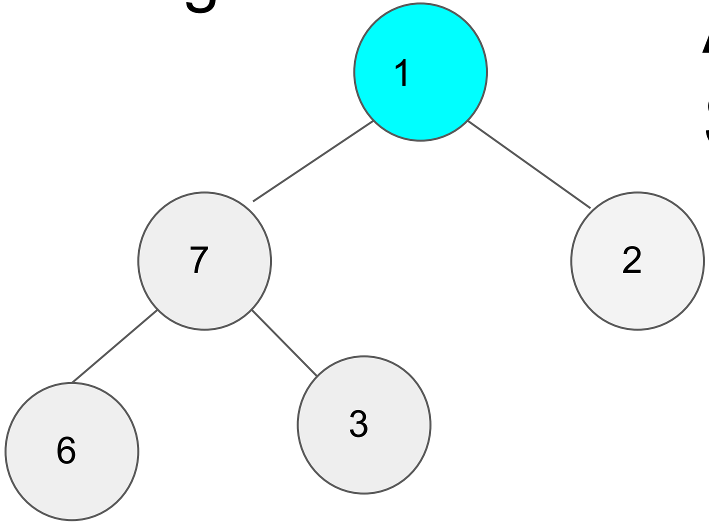
Sift Down the Root



Array: [8, 7, 2, 6, 3, 1]

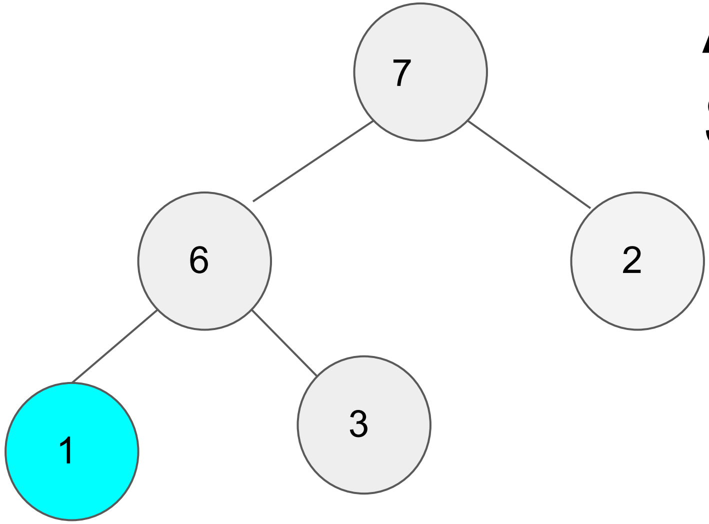
Sorted region: [9, 10]

Replace the root with the last element and add root to sorted region



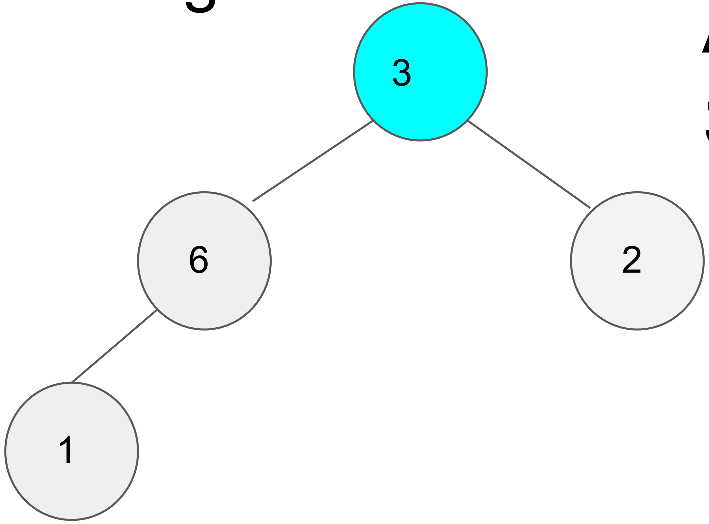
Array: [1, 7, 2, 6, 3]
Sorted region: [8, 9, 10]

Sift Down the Root



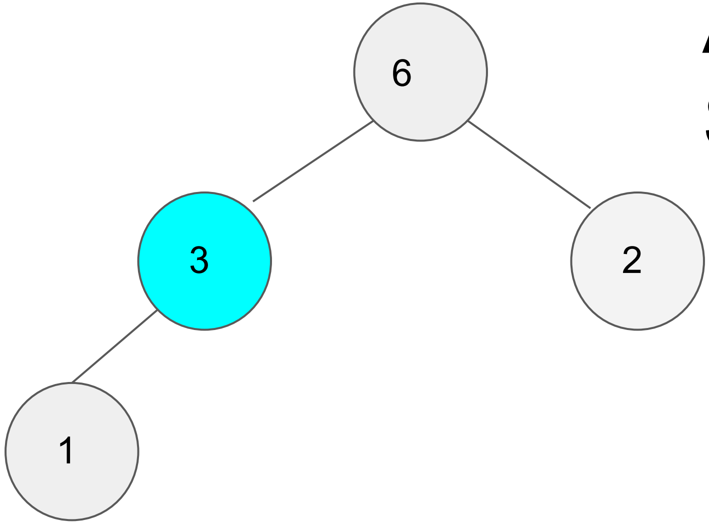
Array: [7, 6, 2, 1, 3]
Sorted region: [8, 9, 10]

Replace the root with the last element and add root to sorted region



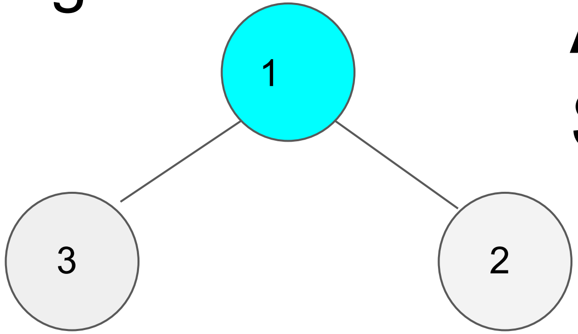
Array: [3, 6, 2, 1]
Sorted region: [7, 8, 9, 10]

Sift Down the Root



Array: [6, 3, 2, 1]
Sorted region: [7, 8, 9, 10]

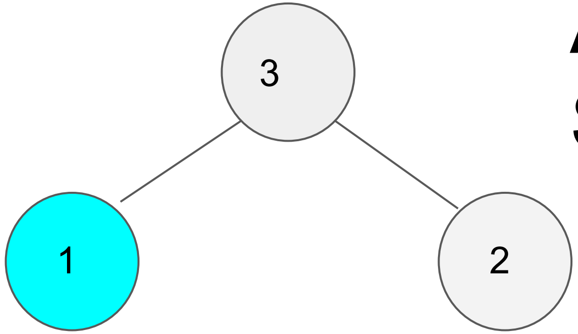
Replace the root with the last element and add root to sorted region



Array: [1, 3, 2]

Sorted region: [6, 7, 8, 9, 10]

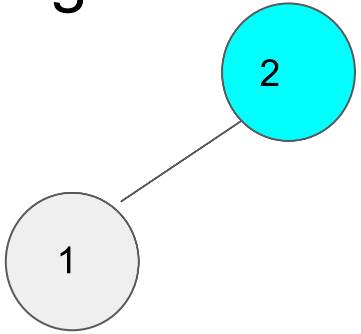
Sift Down the Root



Array: [3, 1, 2]

Sorted region: [6, 7, 8, 9, 10]

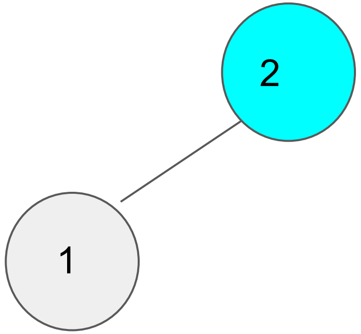
Replace the root with the last element and add root to sorted region



Array: [2, 1]

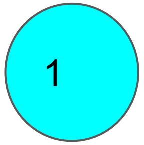
Sorted region: [3, 6, 7, 8, 9, 10]

Sift Down the Root



Array: [2, 1]
Sorted region: [3, 6, 7, 8, 9, 10]

Replace the root with the last element and add root to sorted region



Array: [1]

Sorted region: [2, 3, 6, 7, 8, 9, 10]

Size is now no longer > 0, add root to sorted region

Sorted region: [1, 2, 3, 6, 7, 8, 9, 10]

Complexity: O(n log n)

Reason: n calls to siftDown(), which at most iterates through the height of the tree, which is log n.