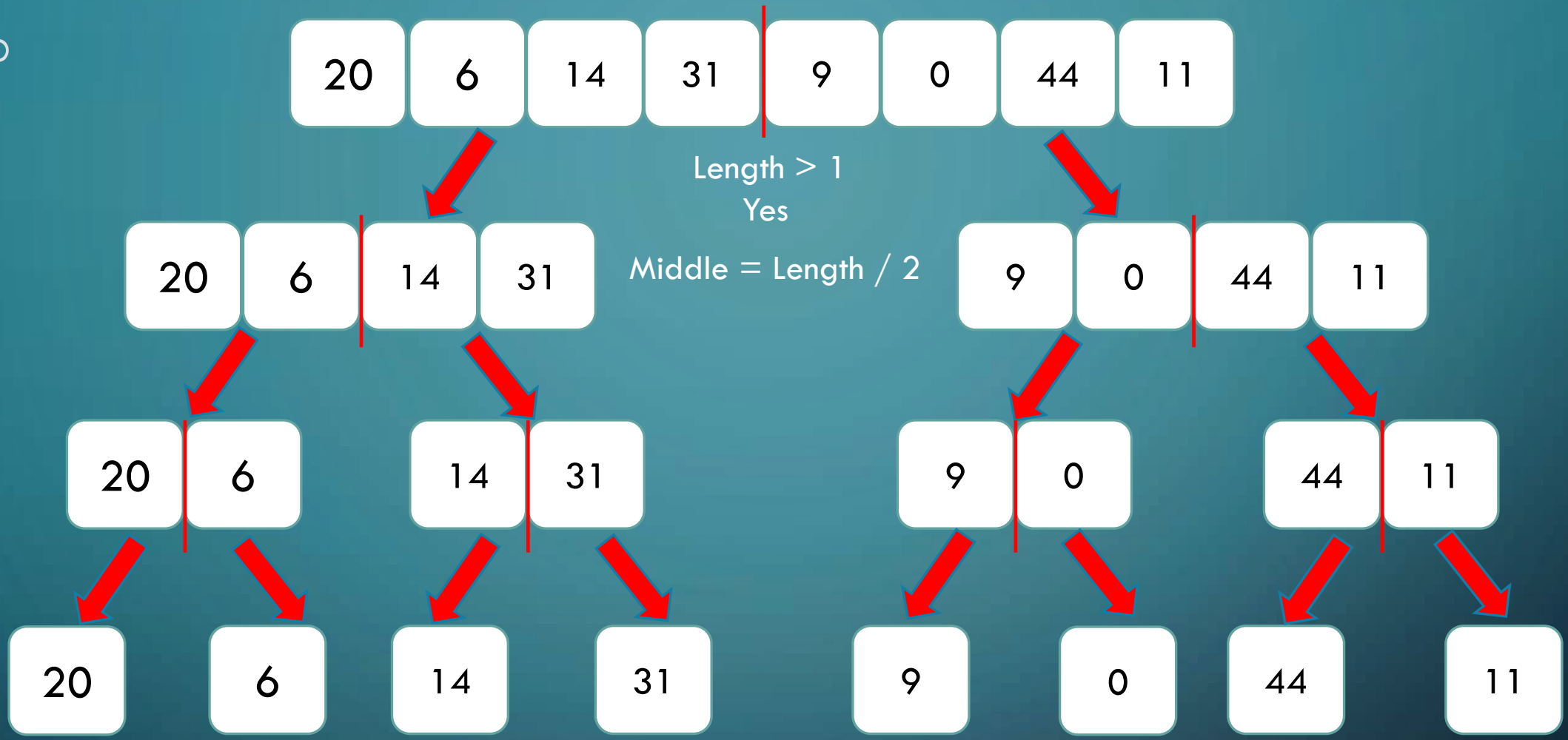
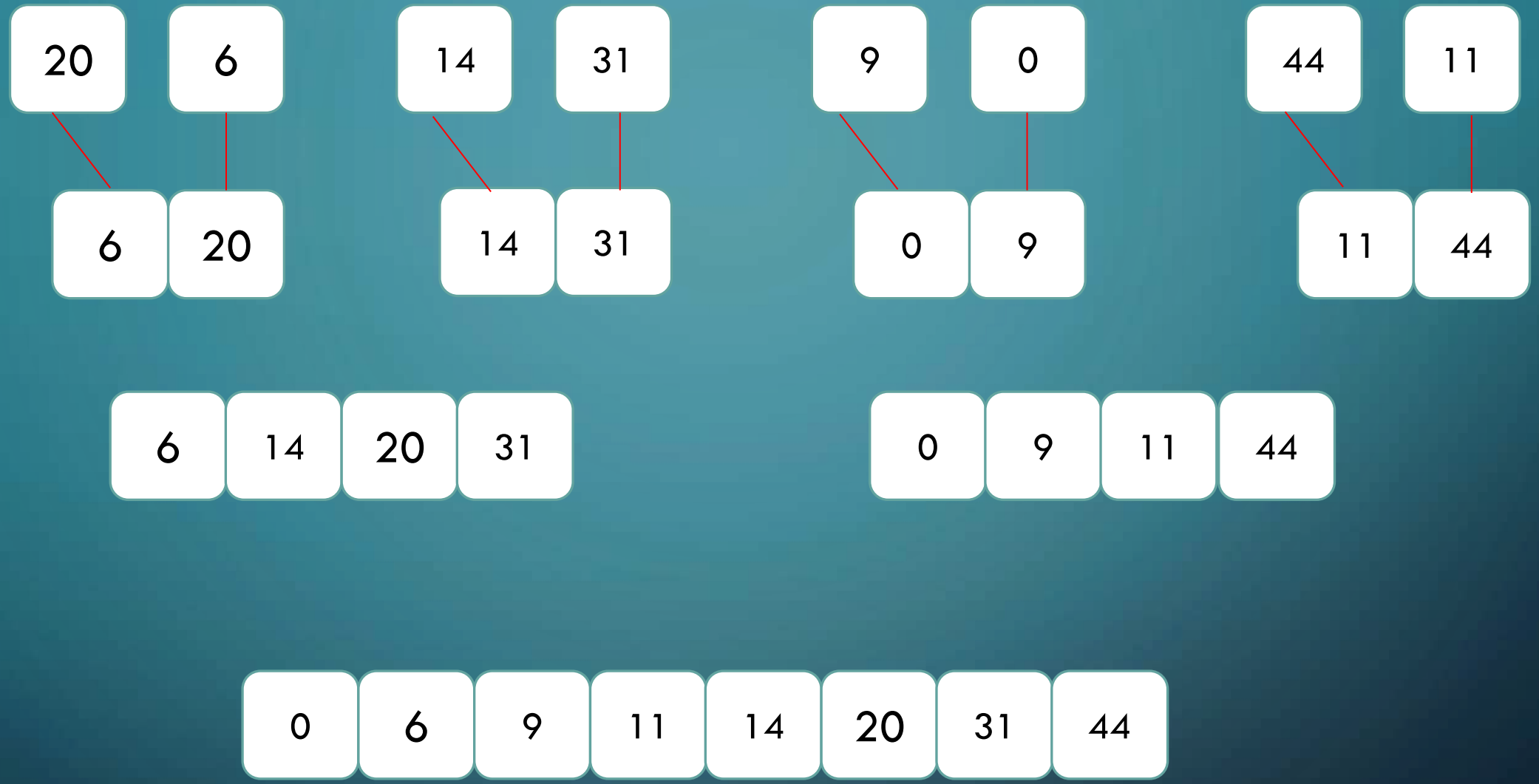
<MERGE SORT & HEAP SORT>

<MAHAN ZIYARI>

MERGE SORT

- ✓ MergeSort is Divide and Conquer algorithm.
- ✓ It divides input array in two halves, calls it self for the two halves and then merges the two sorted halves.
- ✓ Two functions are involved in this algorithm:
  - The merge() function is used for merging two halves.
  - The MergeSort() recursively calls itself to divide the array till size becomes one.


20 6 14 31 9 0 44 11
Length > 1
Yes
Middle = Length / 2
20 6 14 31
9 0 44 11
20 6
14 31
9 0
44 11
20
6
14
31
9
0
44
11


20
6
14
31
9
0
44
11
6
20
14
31
0
9
11
44
6
14
20
31
0
9
11
44
0
6
9
11
14
20
31
44

LET'S TAKE A LOOK AT CODE

```
14
15          public static void mergeSort(int array[], int length){
16              // return condition
17              if (length > 1){
18
19                  // declaring middle position and Sub-arrays
20                  int middle = length/ 2;
21                  int[] leftArray = new int[middle];
22                  int[] rightArray = new int[length - middle];
23
24                  // assigning sub-arrays
25                  for (int i = 0; i < middle; i++)
26                      leftArray[i] = array[i];
27                  for (int j = middle; j < length; j++)
28                      rightArray[j - middle] = array[j];
29
30                  // method calls itself for both sub-arrays to divide them
31                  mergeSort(leftArray, middle);
32                  mergeSort(rightArray,  length: length - middle);
33
34                  //passing original array and sub-arrays to merge
35                  merge(array, leftArray, rightArray, middle,  n2: length - middle);
36              }
```

```java
    public static void merge(int[] array, int[] leftArray, int[] rightArray, int n1, int n2){
        int i = 0, j = 0, k = 0;
        /* each time this method is called,
        it compares left-array values with the right one
        and put them in their correct position int the original array*/
        while (i < n1 && j < n2){
            if (leftArray[i] <= rightArray[j])
                array[k++] = leftArray[i++];
            else
                array[k++] = rightArray[j++];
        }

        /* put the remaining values from the sub-array
        which it's pointer doesn't reach the end of array*/
        while (i < n1)
            array[k++] = leftArray[i++];
        while (j < n2)
            array[k++] = rightArray[j++];

    }
}
```

MERGE SORT FEATURES

- Merge sort can work well on any type of data sets irrespective of its size.
- Merge sort is not in place because it requires additional memory space to store the auxiliary arrays.
- The merge sort is external sorting method in which the data that is to be sorted cannot be accommodated in the memory and needed auxiliary memory for sorting.
- Merge sort is stable as two elements with equal value appear in the same order in sorted output as they were in the input unsorted array.

TIME COMPLEXITY

- Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation : $T(n) = 2T(n/2) + O(n)$
  - $2T(n/2)$ is the required time to sort sub-arrays
  - $O(n)$ is the required time to merge the array
- Time complexity of Merge Sort is $O(n \log n)$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.

DID YOU KNOW?

Java 6's `Arrays.sort` method uses Quicksort for arrays of primitives and merge sort for arrays of objects.

RESOURCES THAT USED FOR MERGE SORT

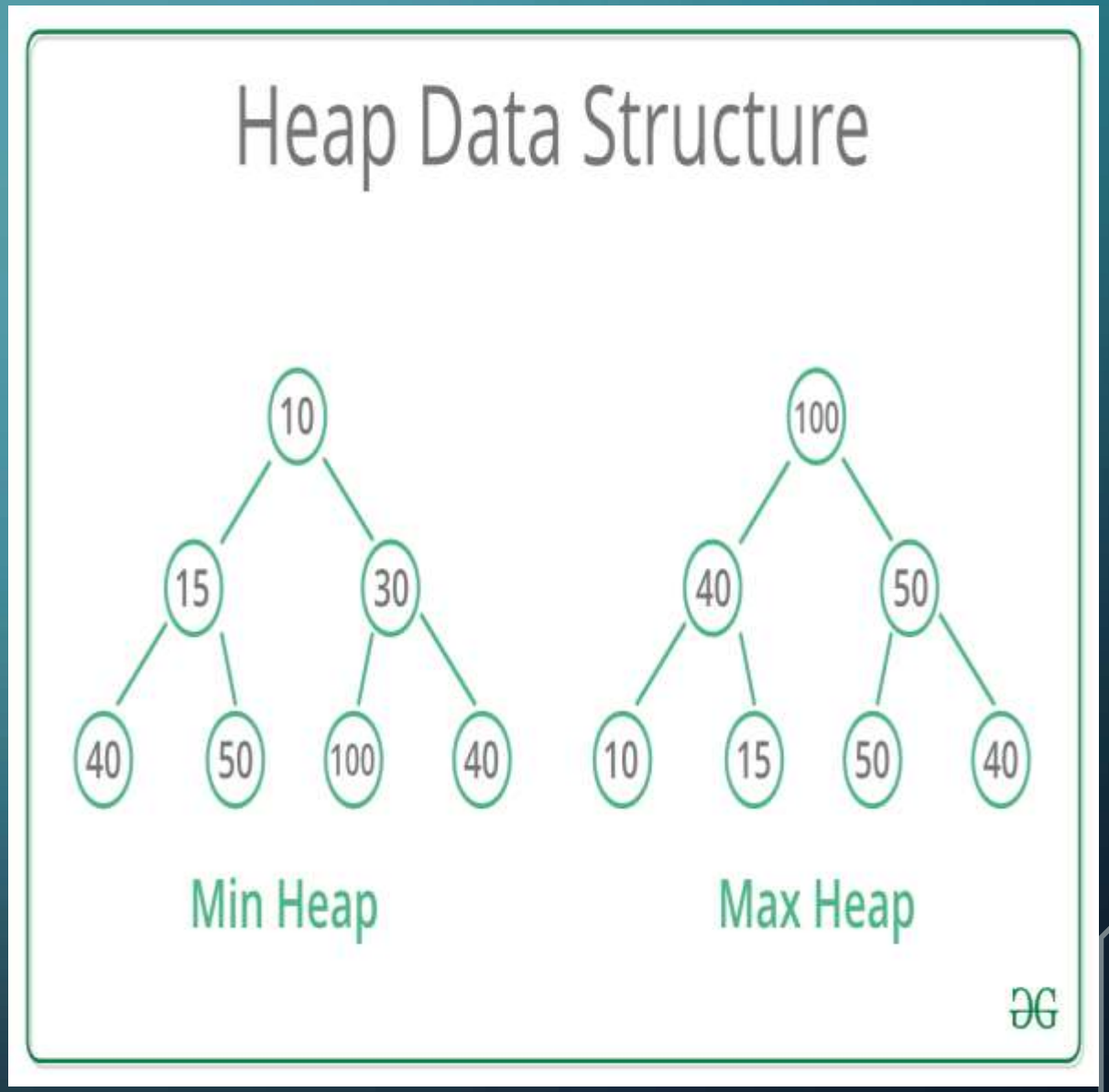
- https://en.wikipedia.org/wiki/Merge_sort
- https://www.geeksforgeeks.org/merge-sort/
- https://www.youtube.com/watch?v=JSceec-wEyw
- https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/
- https://stackoverflow.com/questions/3707190/why-does-javas-arrays-sort-method-use-two-different-sorting-algorithms-for-diff

HEAP SORT

- ✓ In this Algorithm we first build the heap using the given elements.
- ✓ We create a MaxHeap to sort the elements in ascending order.
- ✓ Once the heap is created, we swap the root node with the last node and delete the last node from the heap.

HEAP EXAMPLES

- In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
- In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

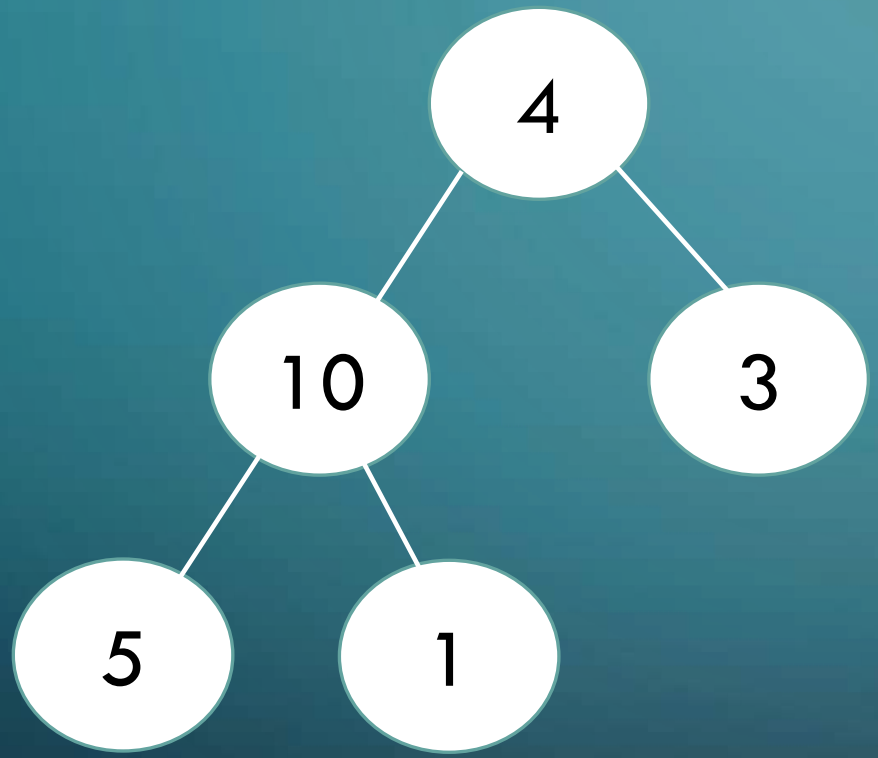


WHY ARRAY BASED REPRESENTATION FOR BINARY HEAP?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

Input Data

4	10	3	5	1



Now that we have build a heap we need to transform it into a max heap

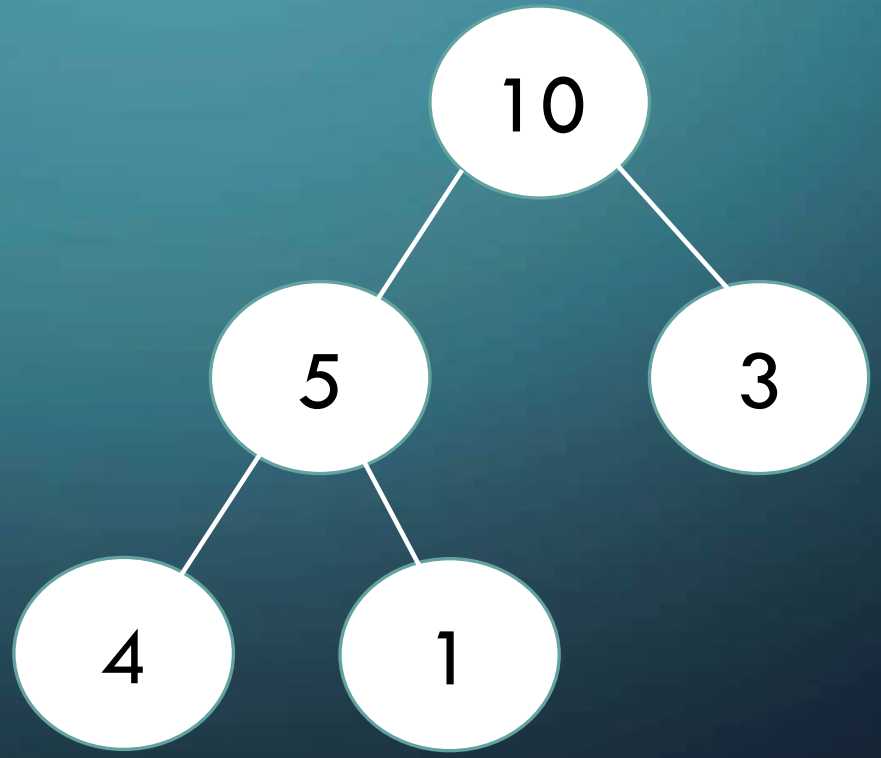
Input Data

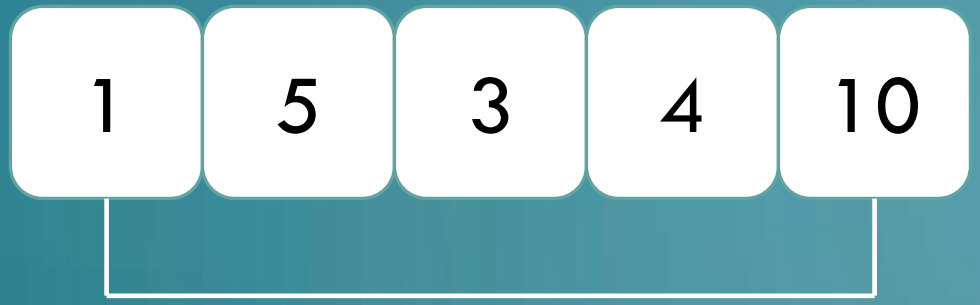
4	10	3	5	1

Rearranged array

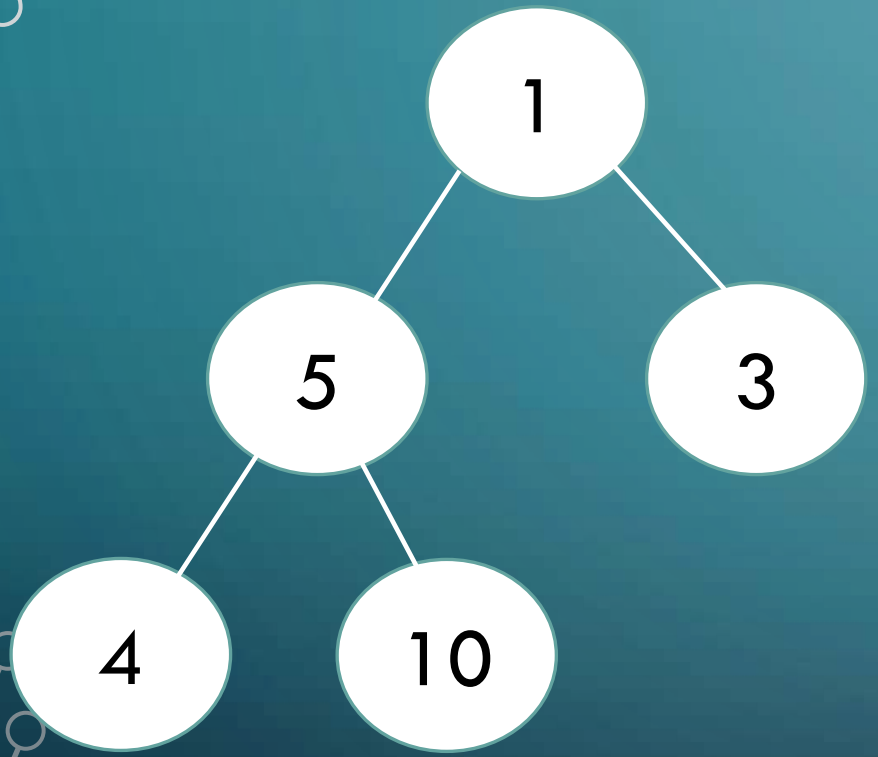
10	5	3	4	1

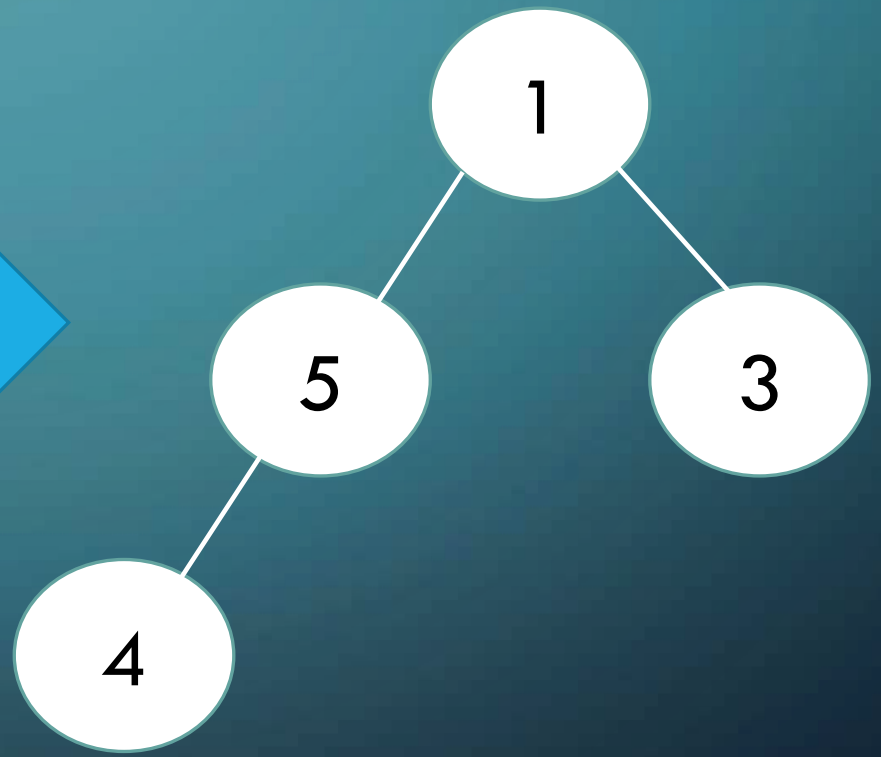
To build a max heap we swap 4 and 10 and then we swap 4 and 5



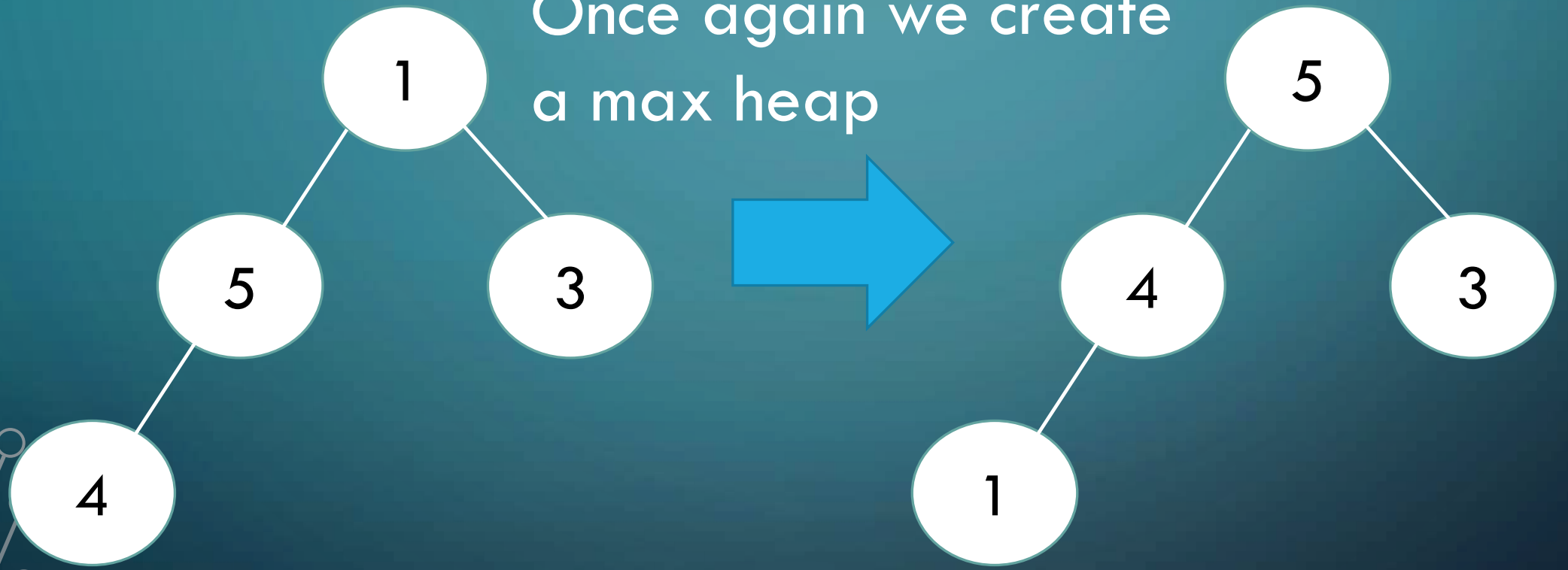


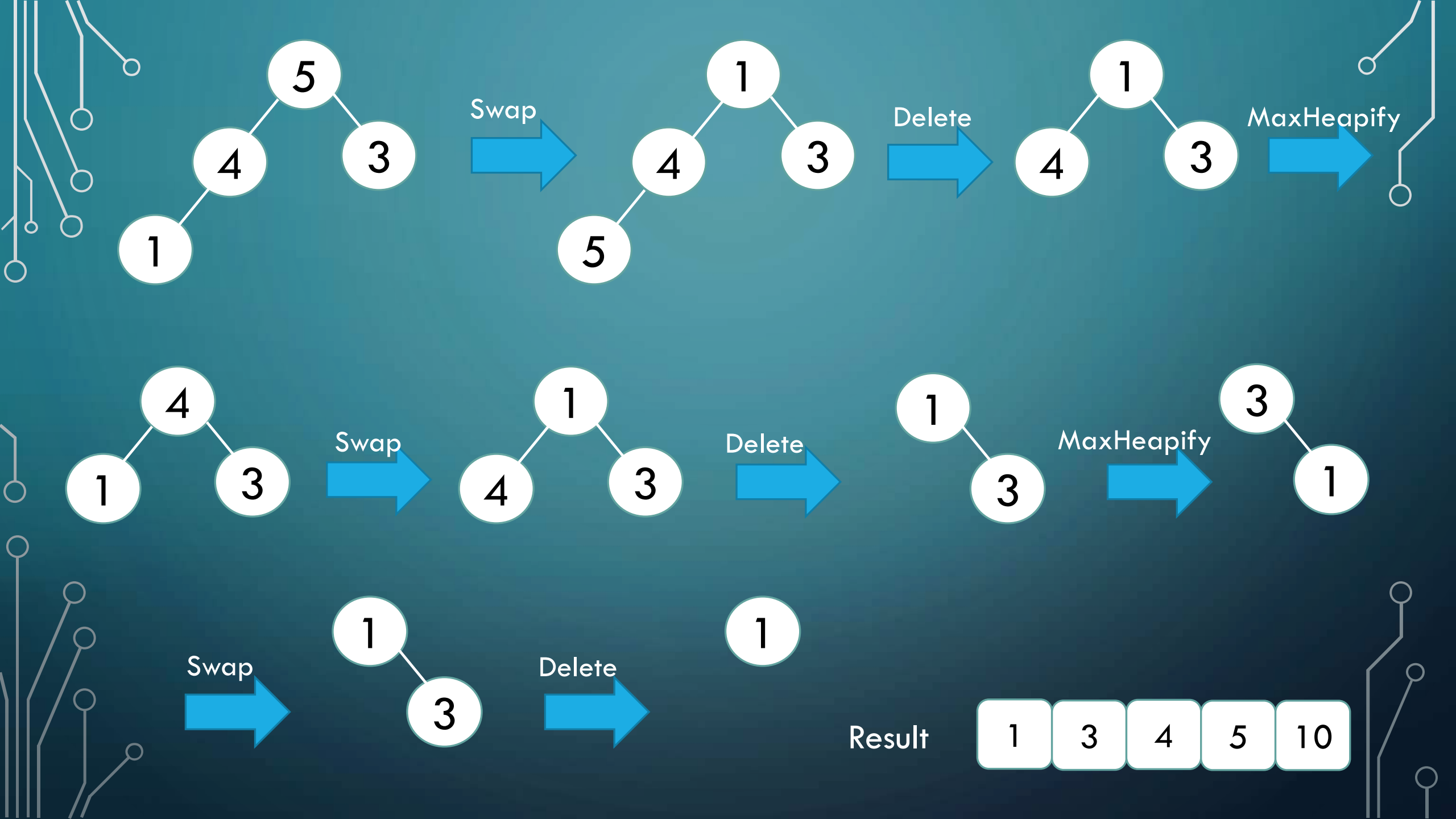
Swap first and last node and delete the last node from heap





Once again we create a max heap




5
4
3
1
Swap
1
4
3
5
Delete
1
4
3
MaxHeapify
4
1
3
Swap
1
4
3
Delete
1
3
MaxHeapify
3
1
Swap
1
3
Delete
1
Result
1
3
4
5
10

NOW LET'S TAKE A LOOK AT CODE

```java
public static void heapSort(int[] array, int length){
    // Build heap (rearrange array)
    for (int i = length / 2 - 1; i >= 0; i--)
        heapify(array, length, i);

    // One by one extract an element from heap
    for (int i=length - 1; i > 0; i--)
    {
        // Move current root to end
        int temp = array[0];
        array[0] = array[i];
        array[i] = temp;

        // call max heapify on the reduced heap
        heapify(array, i, index: 0);
    }
}
```

```java
// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
public static void heapify(int[] array, int length, int index)
{
    int largest = index; // Initialize largest as root
    int l = 2*index + 1; // left = 2*i + 1
    int r = 2*index + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < length && array[l] > array[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < length && array[r] > array[largest])
        largest = r;

    // If largest is not root
    if (largest != index)
    {
        int swap = array[index];
        array[index] = array[largest];
        array[largest] = swap;

        // Recursively heapify the affected sub-tree
        heapify(array, length, largest);
    }
}
```

TIME AND SPACE COMPLEXITY

- The buildMaxHeap() operation is run once, and is O(n) in performance. The siftDown() function is O(log n), and is called n times. Therefore, the performance of this algorithm in all three cases is O(n + n log n) = O(n log n)
- HeapSort is an In-place algorithm, so the space complexity of this sort is O(1)

NOTES

- Heap sort is an in-place algorithm.
- Its typical implementation is not stable, but can be made stable.
- Heap sort algorithm has limited uses because QuickSort and MergeSort are better in practice. Nevertheless, the Heap data structure itself is enormously used.

RESOURCES THAT USED FOR HEAP SORT

- https://en.wikipedia.org/wiki/Heapsort
- https://www.geeksforgeeks.org/heap-data-structure/
- https://www.geeksforgeeks.org/heap-sort/
- https://www.youtube.com/watch?v=MtQL_ll5KhQ&list=PLqM7alHXFySHrGIxeBOo4-mKO4H8j2knW&index=2&t=0s

MANY THANKS FOR YOUR ATTENTION