CmSc 250 Intro to Algorithms

Sorting Algorithms
Heapsort

1. Heapsort

Based on Priority Queues
Sorts in O(NlogN) time by performing N times deleteMin operations.

Why O(NlogN)?

- Each deleteMin operation takes logN running time.
- N times performing deleteMin → NlogN running time

Basic algorithm (not very efficient - extra memory required):

1. Store N elements in a binary heap tree.
2. Perform deleteMin operation N times, storing each element deleted from the heap into another array.
3. Copy back the array.

Improvements: Use the same array to store the deleted elements instead of using another array

How can we do this?

> After each deletion we percolate down the hole that is formed after the deletion, and as a result we get a vacant position in the array - the last cell. There we store the deleted element.

When all the elements are deleted and stored in the same array following the above method, the elements will be there in reversed order.

What is the remedy for this?

> Store the elements in the binary heap tree in reverse order of priority - then at the end the elements in the array will be in correct order.

Complexity of heap sort: **O(NlogN)** is the rough estimate.

Here is an example of heapsort.

Learning Goals

- Be able to explain how heapsort works and its complexity.

Exam-like questions

1. Explain the complexity of heapsort
2. Given an array e.g. 17, 23, 10, 1, 7, 16, 9, 20, sort it on paper using heapsort
   Write down explicitly each step.

Back to Contents page

Created by Lydia Sinapova

An Example of Heapsort:

Given an array of 6 elements: 15, 19, 10, 7, 17, 16, sort it in ascending order using heap sort.

Steps:

1. Consider the values of the elements as priorities and build the heap tree.
2. Start deleteMin operations, storing each deleted element at the end of the heap array.

After performing step 2, the order of the elements will be opposite to the order in the heap tree. Hence, if we want the elements to be sorted in ascending order, we need to build the heap tree in descending order - the greatest element will have the highest priority.

Note that we use only one array , treating its parts differently:

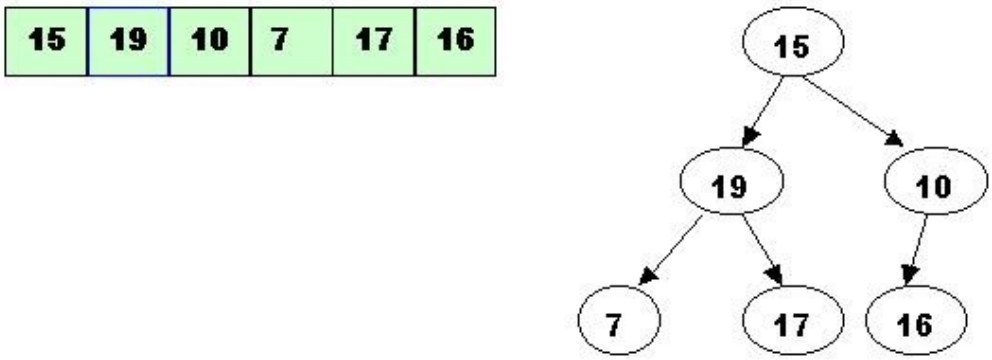
a. when building the heap tree, part of the array will be considered as the heap, and the rest part - the original array.
b. when sorting, part of the array will be the heap, and the rest part - the sorted array.

This will be indicated by colors: white for the original array, blue for the heap and red for the sorted array

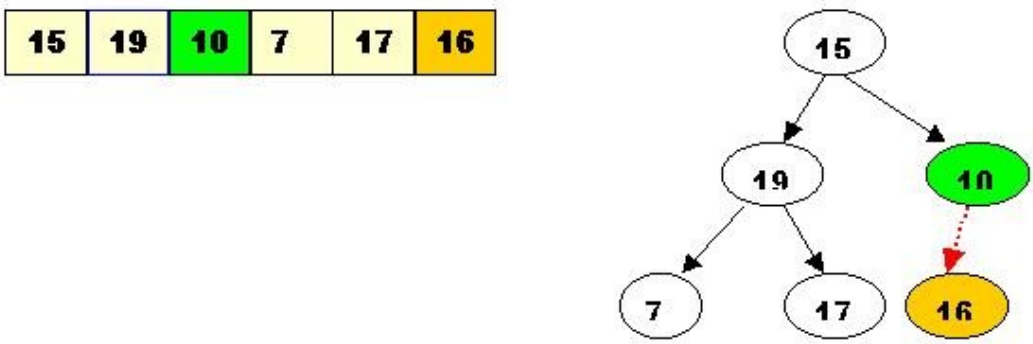
Here is the array: 15, 19, 10, 7, 17, 6

A. Building the heap tree

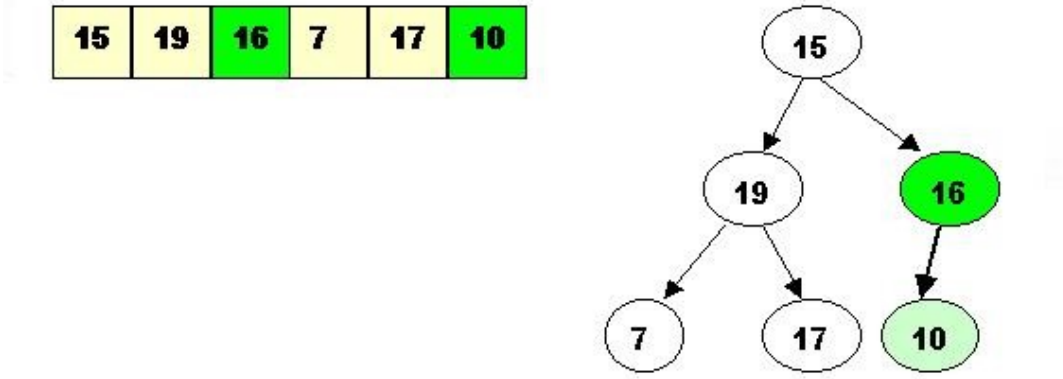
The array represented as a tree, complete but not ordered:



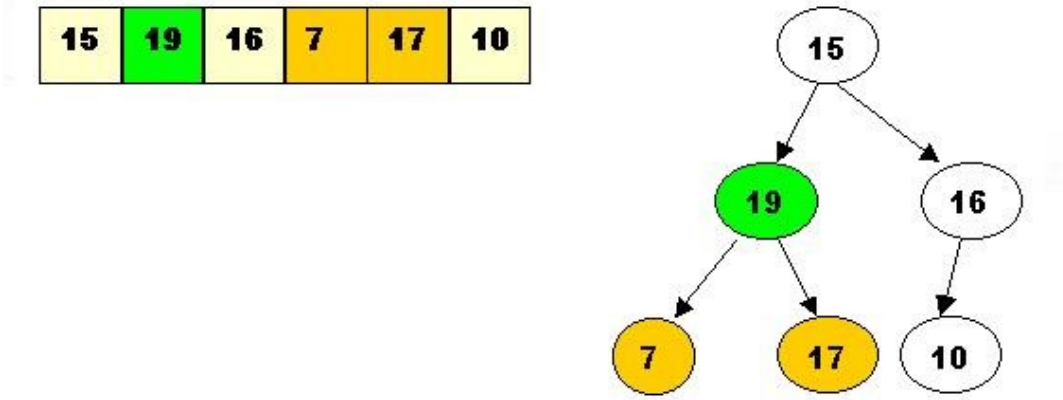
Start with the rightmost node at height 1 - the node at position 3 = Size/2.
It has one greater child and has to be percolated down:



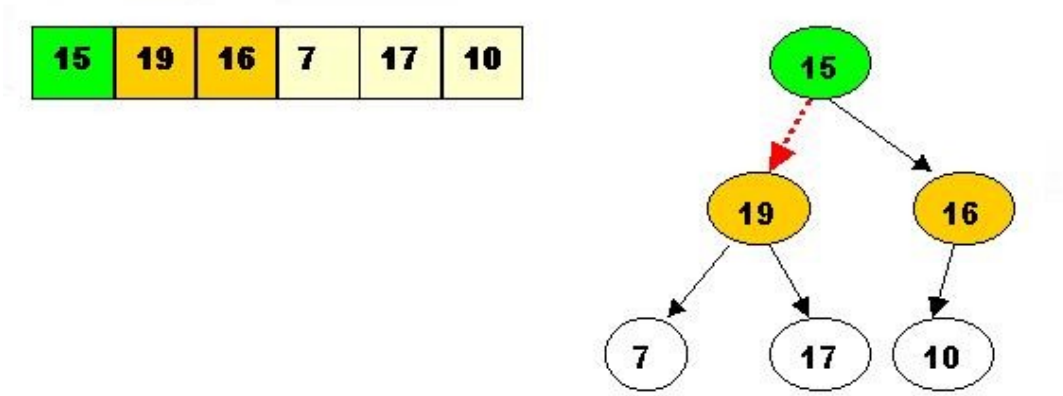
After processing array[3] the situation is:



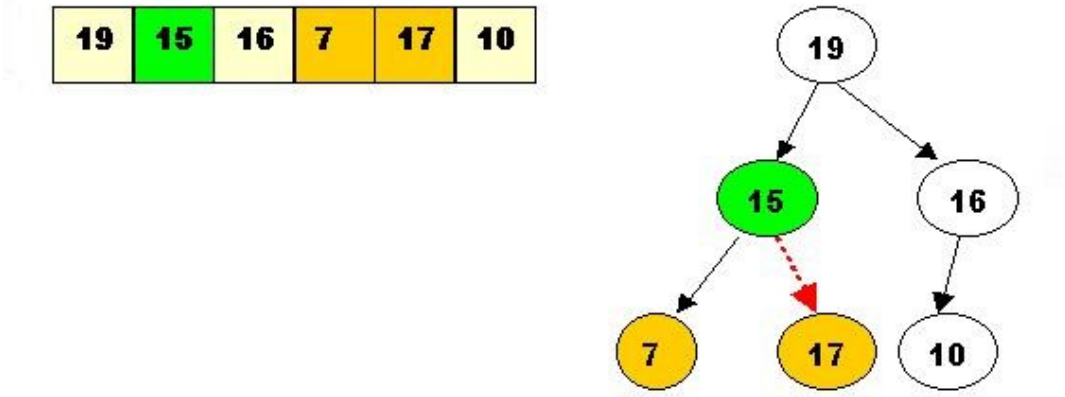
Next comes array[2]. Its children are smaller, so no percolation is needed.



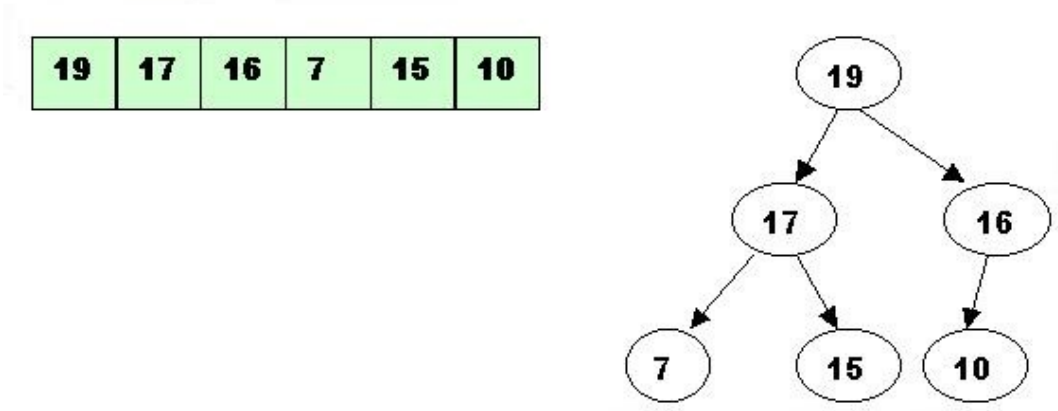
The last node to be processed is array[1]. Its left child is the greater of the children.
The item at array[1] has to be percolated down to the left, swapped with array[2].



As a result the situation is:



The children of array[2] are greater, and item 15 has to be moved down further, swapped with array[5].

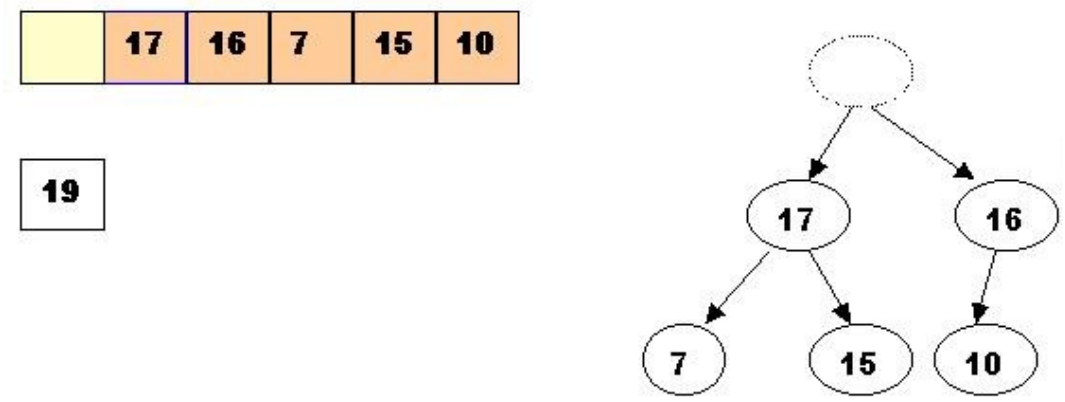


Now the tree is ordered, and the binary heap is built.

B. Sorting - performing deleteMax operations:

1. Delete the top element 19.

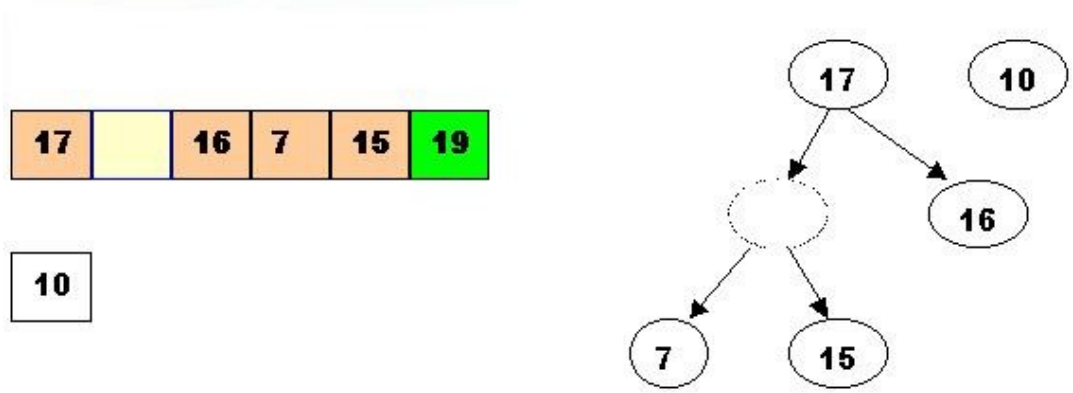
1.1. Store 19 in a temporary place. A hole is created at the top



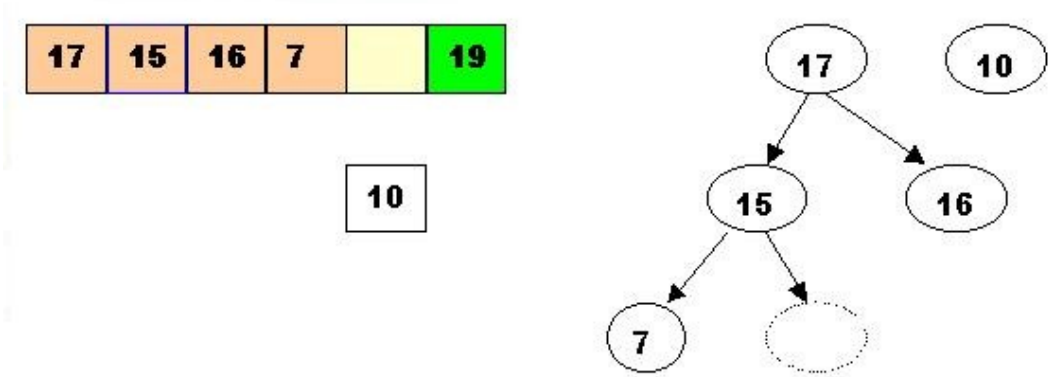
1.2. Swap 19 with the last element of the heap.

As 10 will be adjusted in the heap, its cell will no longer be a part of the heap.
Instead it becomes a cell from the sorted array

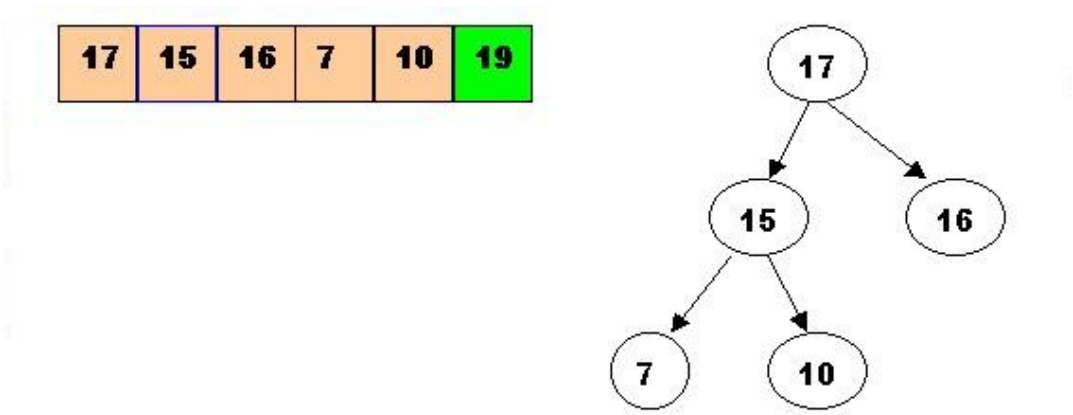
1.3. Percolate down the hole



1.4. Percolate once more (10 is less that 15, so it cannot be inserted in the previous hole)

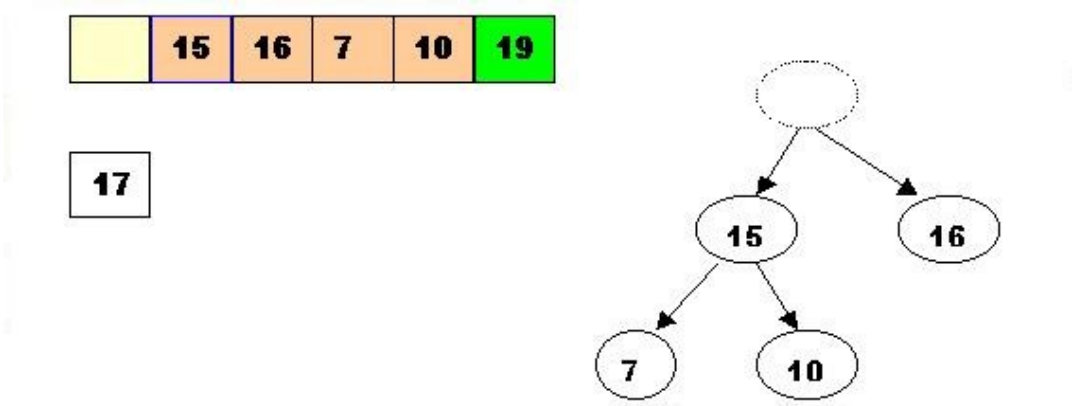


Now 10 can be inserted in the hole



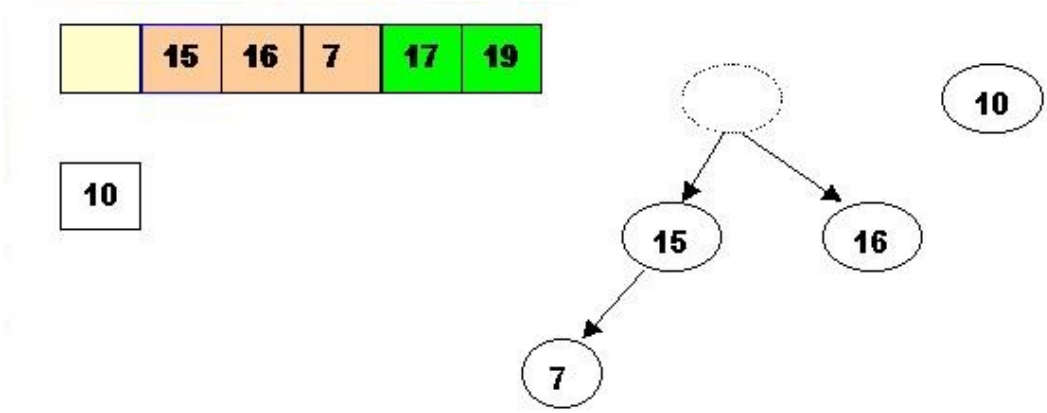
2. DeleteMax the top element 17

2.1. Store 17 in a temporary place. A hole is created at the top

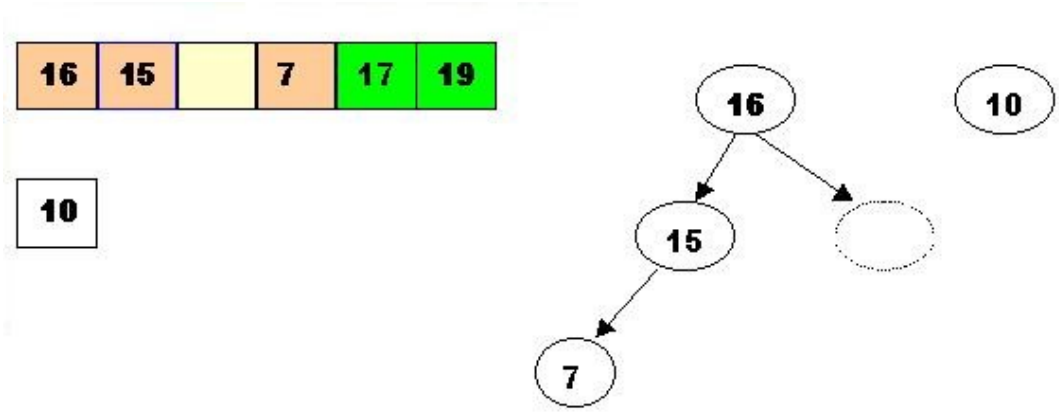


2.2. Swap 17 with the last element of the heap.

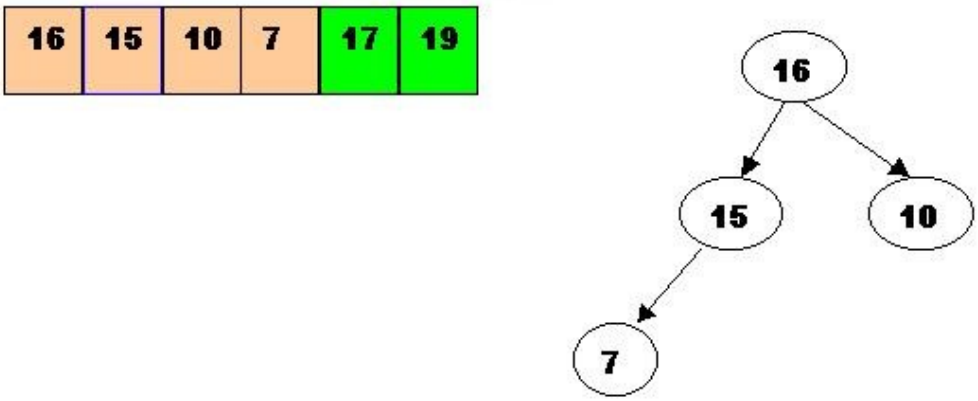
As 10 will be adjusted in the heap, its cell will no longer be a part of the heap.
Instead it becomes a cell from the sorted array



2.3. The element 10 is less than the children of the hole, and we percolate the hole down:

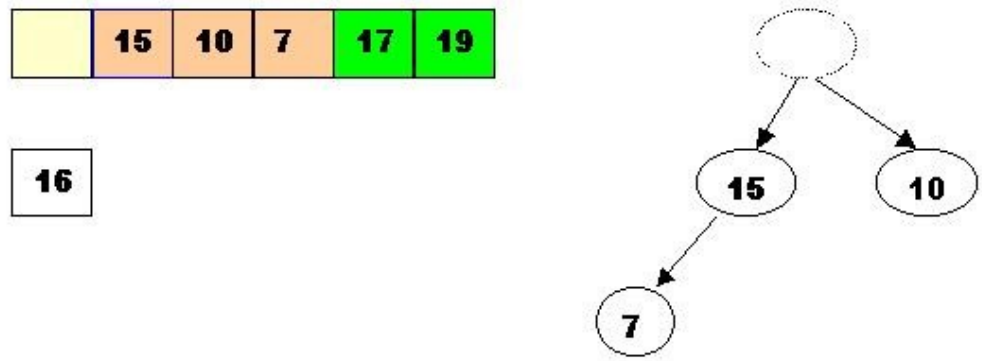


2.4. Insert 10 in the hole



3. DeleteMax 16

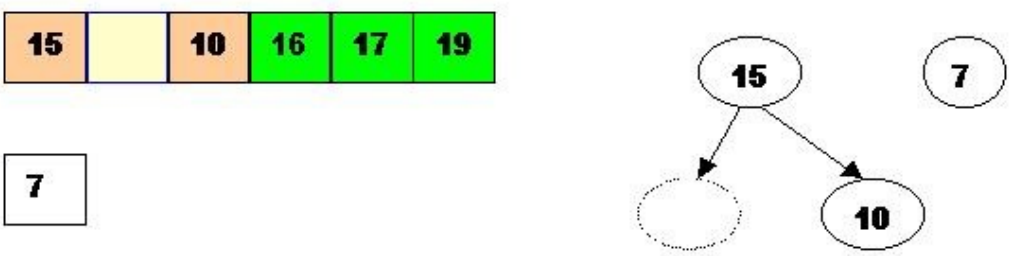
3.1. Store 16 in a temporary place. A hole is created at the top



3.2. Swap 16 with the last element of the heap.

As 7 will be adjusted in the heap, its cell will no longer be a part of the heap.
Instead it becomes a cell from the sorted array

3.3. Percolate the hole down (7 cannot be inserted there - it is less than the children of the hole)



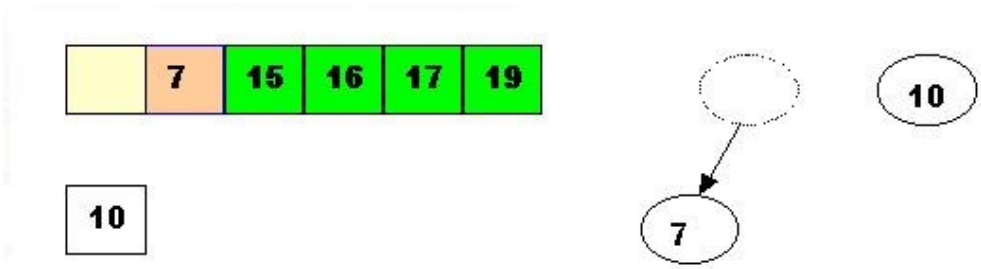
3.4. Insert 7 in the hole

4. DeleteMax the top element 15

4.1. Store 15 in a temporary location. A hole is created.

4.2. Swap 15 with the last element of the heap.

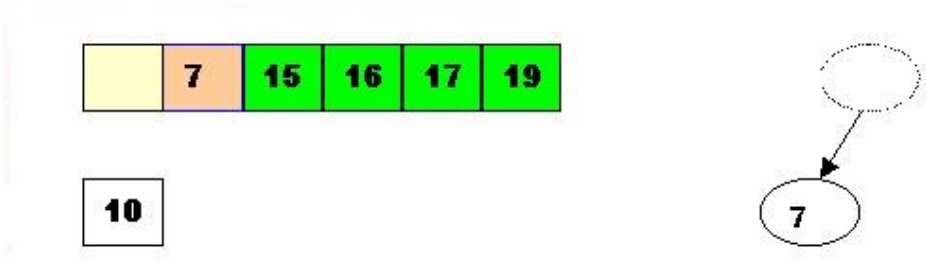
As 10 will be adjusted in the heap, its cell will no longer be a part of the heap.
Instead it becomes a position from the sorted array



4.3. Store 10 in the hole (10 is greater than the children of the hole)

5. DeleteMax the top element 10.

5.1. Remove 10 from the heap and store it into a temporary location.



5.2. Swap 10 with the last element of the heap.

As 7 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array

5.3. Store 7 in the hole (as the only remaining element in the heap

7 is the last element from the heap, so now the array is sorted

Back to HeapSort