# Divide & Conquer Approach (Part-II)

Md. Tanvir Rahman

Lecturer, Dept. of ICT

Mawlana Bhashani Science and Technology University

## Divide and Conquer

- Divide the problem into a number of subproblem.
- Conquer (solve) the sub-problems **recursively**
- Combine (merge) solutions to subproblems into a solution to the original problem

If you want to learn more, watch this: https://www.youtube.com/watch?v=2Rr2tW9zvRg

# Divide and Conquer Algorithms

- Example
  - Binary Search (Discussed in Part-I)
  - Merge Sort (Discussed in Part-I)
  - Quick Sort
  - Heap Sort
  - A lot more! Find yourself!

## Quick Sort

## Quick Sort

- Fastest known sorting algorithm in practice
- Average case: O(N log N) (we don't prove it)
- Worst case: $O(N^2)$
  - But, the worst case seldom happens.
- Another divide-and-conquer recursive algorithm, like merge sort

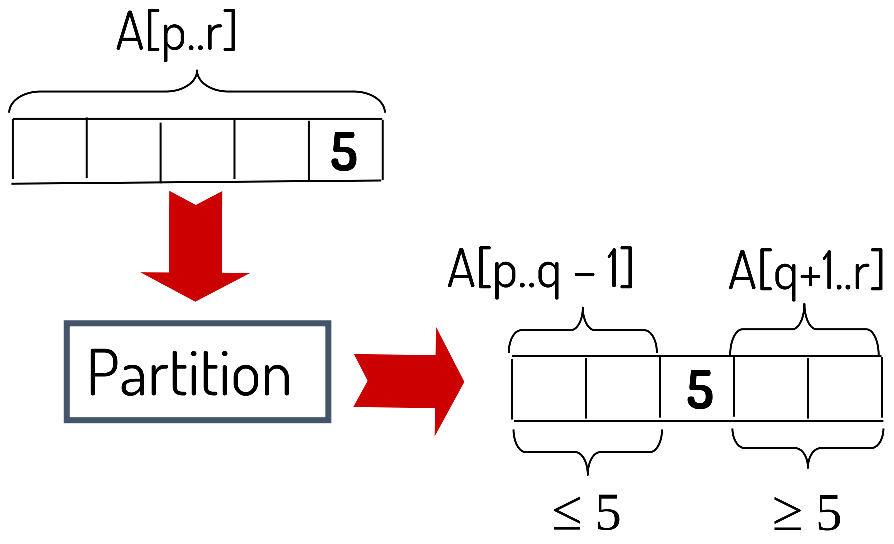
## Design

- Follows the **divide-and-conquer** paradigm.
- ***Divide*** **:** Partition (separate) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$.
    - Each element in $A[p..q-1] \leq A[q]$.
    - $A[q] \leq$ each element in $A[q+1..r]$.
    - Index $q$ is computed as part of the partitioning procedure.
- ***Conquer*** **:** Sort the two subarrays by recursive calls to quicksort.
- ***Combine*** **:** The subarrays are sorted in place – no work is needed to combine them.

# Pseudocode

```
Quicksort(A, p, r)
    if p < r then
        q := Partition(A, p, r);
        Quicksort(A, p, q – 1);
        Quicksort(A, q + 1, r)
    fi
```

```
Partition(A, p, r)
    x := A[r], i := p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```



# Example

**initially:**

```
   p                          r
   2  5  8  3  9  4  1  7  10 6
i  j
```

**note:** pivot (x) = 6

**next iteration:**

```
   2  5  8  3  9  4  1  7  10 6
   i  j
```

**next iteration:**

```
   2  5  8  3  9  4  1  7  10 6
      i  j
```

**next iteration:**

```
   2  5  8  3  9  4  1  7  10 6
      i     j
```

**next iteration:**

```
   2  5  3  8  9  4  1  7  10 6
         i     j
```

```
Partition(A, p, r)
    x := A[r], i := p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
        A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```

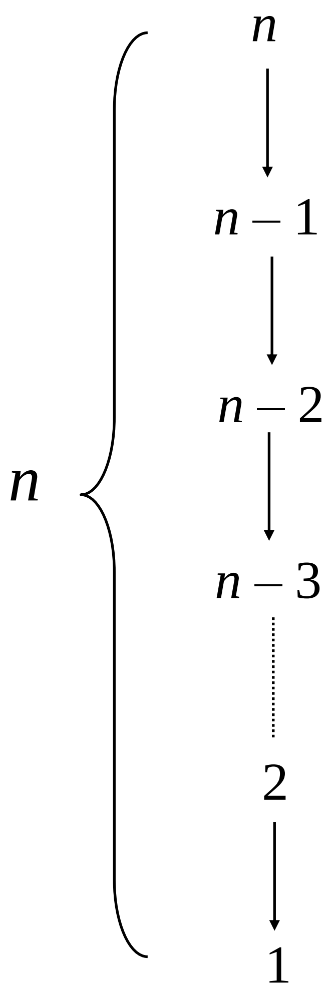
# Example (Continued)

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **next iteration:** | 2 | 5 | 3 | 8 | 9 | 4 | 1 | 7 | 10 | **6** |
| | | | i | | j | | | | | |
| **next iteration:** | 2 | 5 | 3 | 8 | 9 | 4 | 1 | 7 | 10 | **6** |
| | | | i | | | j | | | | |
| **next iteration:** | 2 | 5 | 3 | 4 | 9 | 8 | 1 | 7 | 10 | **6** |
| | | | | i | | | j | | | |
| **next iteration:** | 2 | 5 | 3 | 4 | 1 | 8 | 9 | 7 | 10 | **6** |
| | | | | | i | | | j | | |
| **next iteration:** | 2 | 5 | 3 | 4 | 1 | 8 | 9 | 7 | 10 | **6** |
| | | | | | i | | | | j | |
| **next iteration:** | 2 | 5 | 3 | 4 | 1 | 8 | 9 | 7 | 10 | **6** |
| | | | | | i | | | | | j |
| **after final swap:** | 2 | 5 | 3 | 4 | 1 | **6** | 9 | 7 | 10 | 8 |
| | | | | | i | | | | | j |

```
Partition(A, p, r)
    x := A[r], i := p − 1;
    for j := p to r − 1 do
        if A[j] ≤ x then
            i := i + 1;
        A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```

# Worst-case Partition Analysis

Recursion tree for worst-case partition



Running time for worst-case partitions at each recursive level:

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \text{PartitionTime}(n) \\
&= T(n-1) + \Theta(n) \\
&= \textstyle\sum_{k=1 \text{ to } n} \Theta(k) \\
&= \Theta(\textstyle\sum_{k=1 \text{ to } n} k) = \Theta(\, n(n+1) / 2 \,) \\
&= \Theta(\, n^2 \,)
\end{aligned}
$$

Ref: https://www.youtube.com/watch?v=4nVbJV5pZa8

## Heap Sort

## Heap

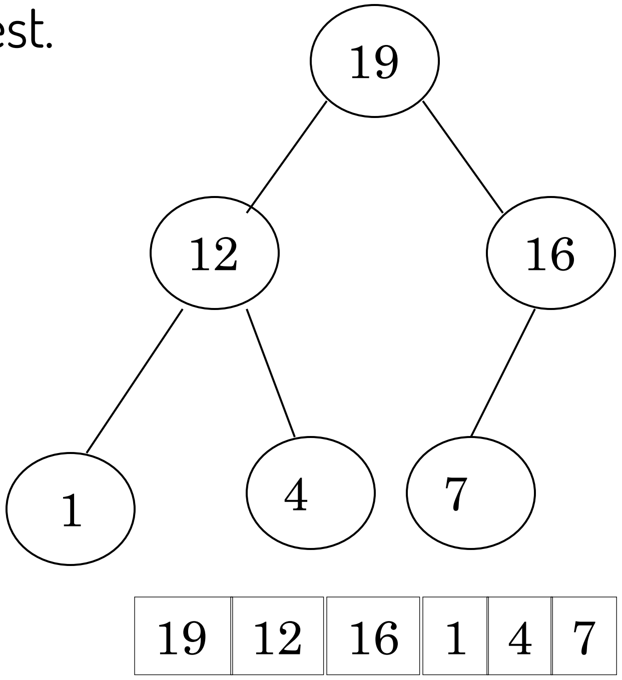
A heap is a data structure that stores a collection of objects (with keys), and has the following properties:

- Complete Binary tree
- Heap Order

It is implemented as an array where each node in the tree corresponds to an element of the array.

## Heap

- The binary heap data structures is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The array is completely filled on all levels except possibly lowest.



Array A

## Heap

- The root of the tree A[1] and given index $i$ of a node, the indices of its parent, left child and right child can be computed

```
PARENT ($i$)
    return floor($i/2$)
LEFT ($i$)
    return 2$i$
RIGHT ($i$)
    return 2$i$+1
```

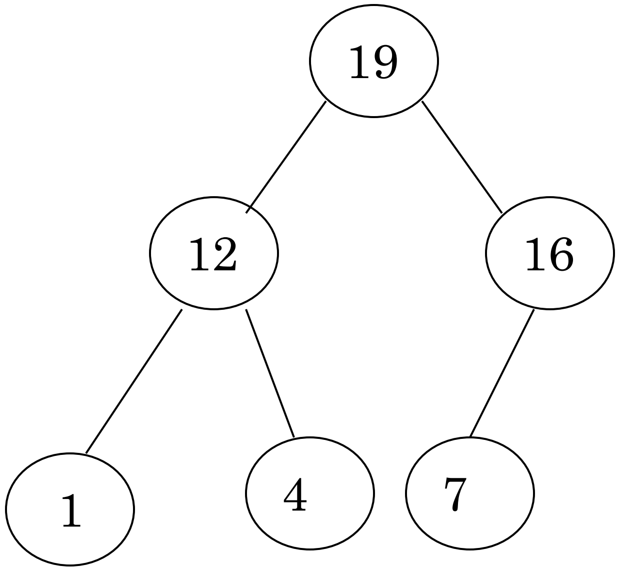
## Definition

- Max Heap
  - Store data in ascending order
  - Has property of

    $A[Parent(i)] \geq A[i]$
- Min Heap
  - Store data in descending order
  - Has property of

    $A[Parent(i)] \leq A[i]$

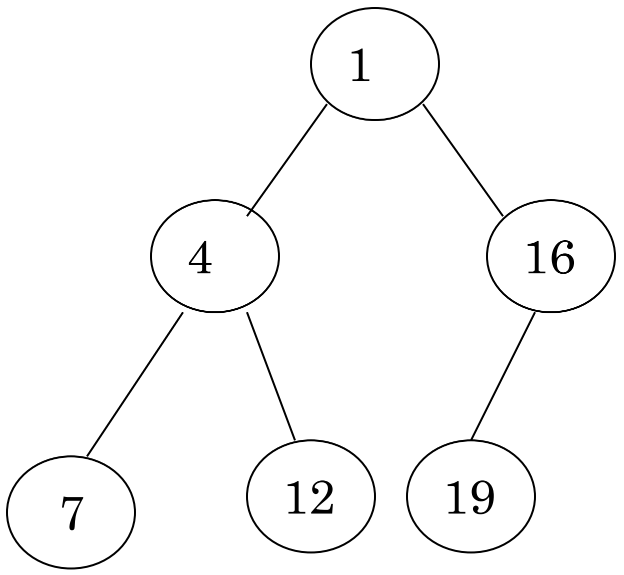
## Max Heap Example



|  |  |  |  |  |  |
|---|---|---|---|---|---|
| 19 | 12 | 16 | 1 | 4 | 7 |

Array A

## Min heap example



| | | | | | |
|---|---|---|---|---|---|
| 1 | 4 | 16 | 7 | 12 | 19 |

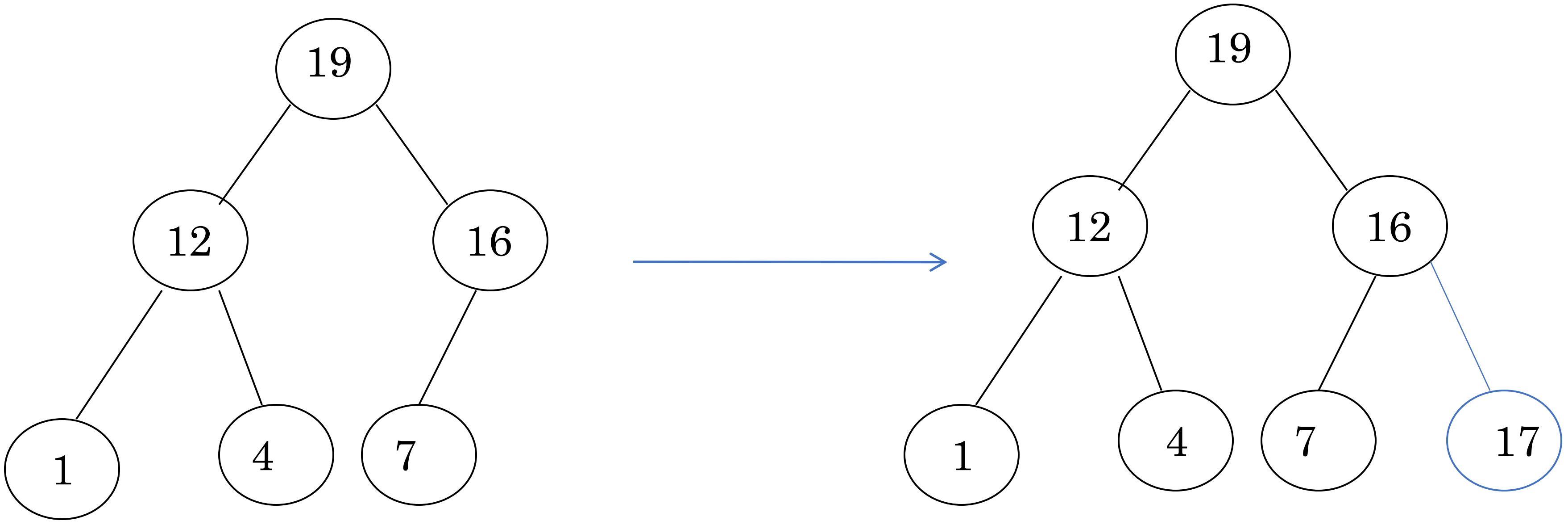
Array A

## Insertion

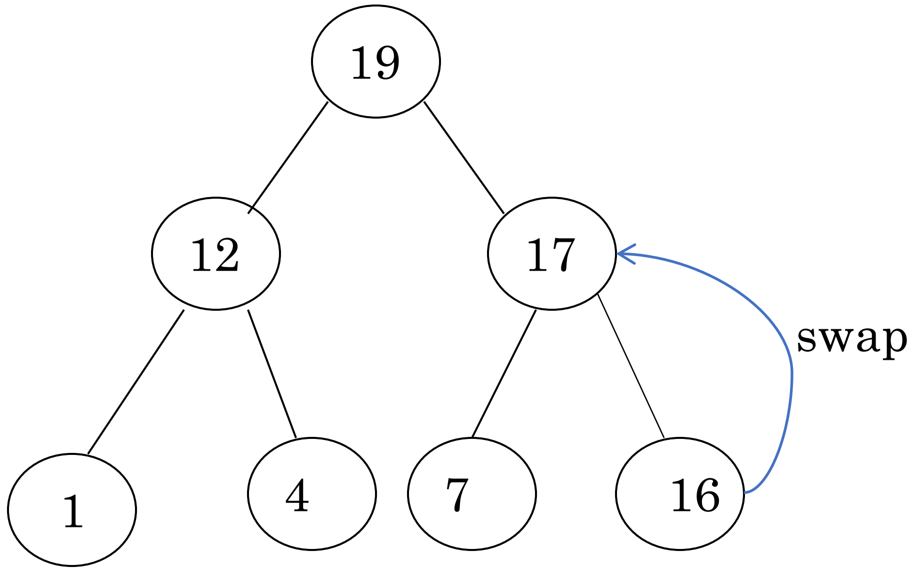
- Algorithm
  1. Add the new element to the next available position at the lowest level
  2. Restore the max-heap property if violated
     - General strategy is percolate up (or bubble up): if the parent of the element is smaller than the element, then interchange the parent and child.

     OR

     Restore the min-heap property if violated
     - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent and child.



Insert 17



Percolate up to maintain the heap property

## Deletion

- Delete max
  - Copy the last number to the root ( overwrite the maximum element stored there ).
  - Restore the max heap property by percolate down.

- Delete min
  - Copy the last number to the root ( overwrite the minimum element stored there ).
  - Restore the min heap property by percolate down.

## Heap Sort

A sorting algorithm that works by first organizing the data to be sorted into a special type of binary tree called a heap

## Procedures on Heap

- Heapify
- Build Heap
- Heap Sort

## Heapify

- Heapify picks the **largest child key** and compare it to the parent key. If parent key is larger than heapify quits, otherwise it swaps the parent key with the largest child key. So that the parent is now becomes larger than its children.

```
Heapify(A, i)
{
    l ← left(i)
    r ← right(i)
    if l <= heapsize[A] and A[l] > A[i]
       then largest ←l
       else largest ← i
    if r <= heapsize[A] and A[r] > A[largest]
       then largest ← r
    if largest != i
       then swap A[i] ←→ A[largest]
          Heapify(A, largest)
}
```

## Build Heap

- We can use the procedure 'Heapify' in a bottom-up fashion to convert an array A[1 . . *n*] into a heap. Since the elements in the subarray A[*n*/2 +1 . . *n*] are all leaves, the procedure BUILD_HEAP goes through the remaining nodes of the tree and runs 'Heapify' on each one. The bottom-up order of processing node guarantees that the subtree rooted at children are heap before 'Heapify' is run at their parent.

```
 Buildheap(A)
{
      heapsize[A] ←length[A]
      for i ←llength[A]/2  //down to 1
         do Heapify(A, i)
 }
```

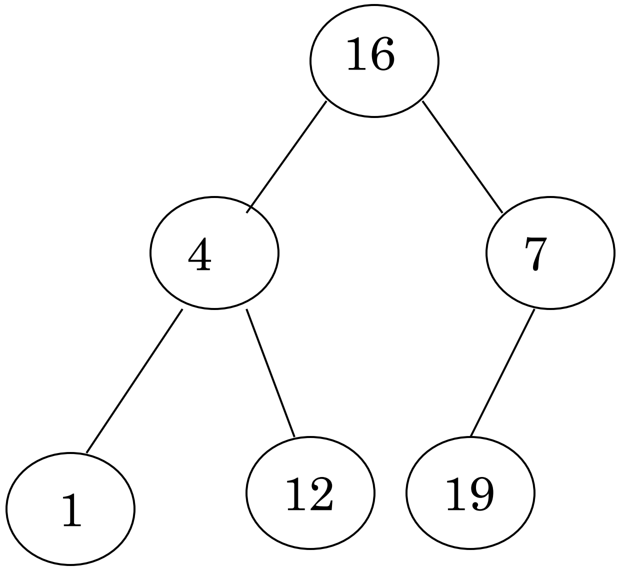
## Heap Sort Algorithm

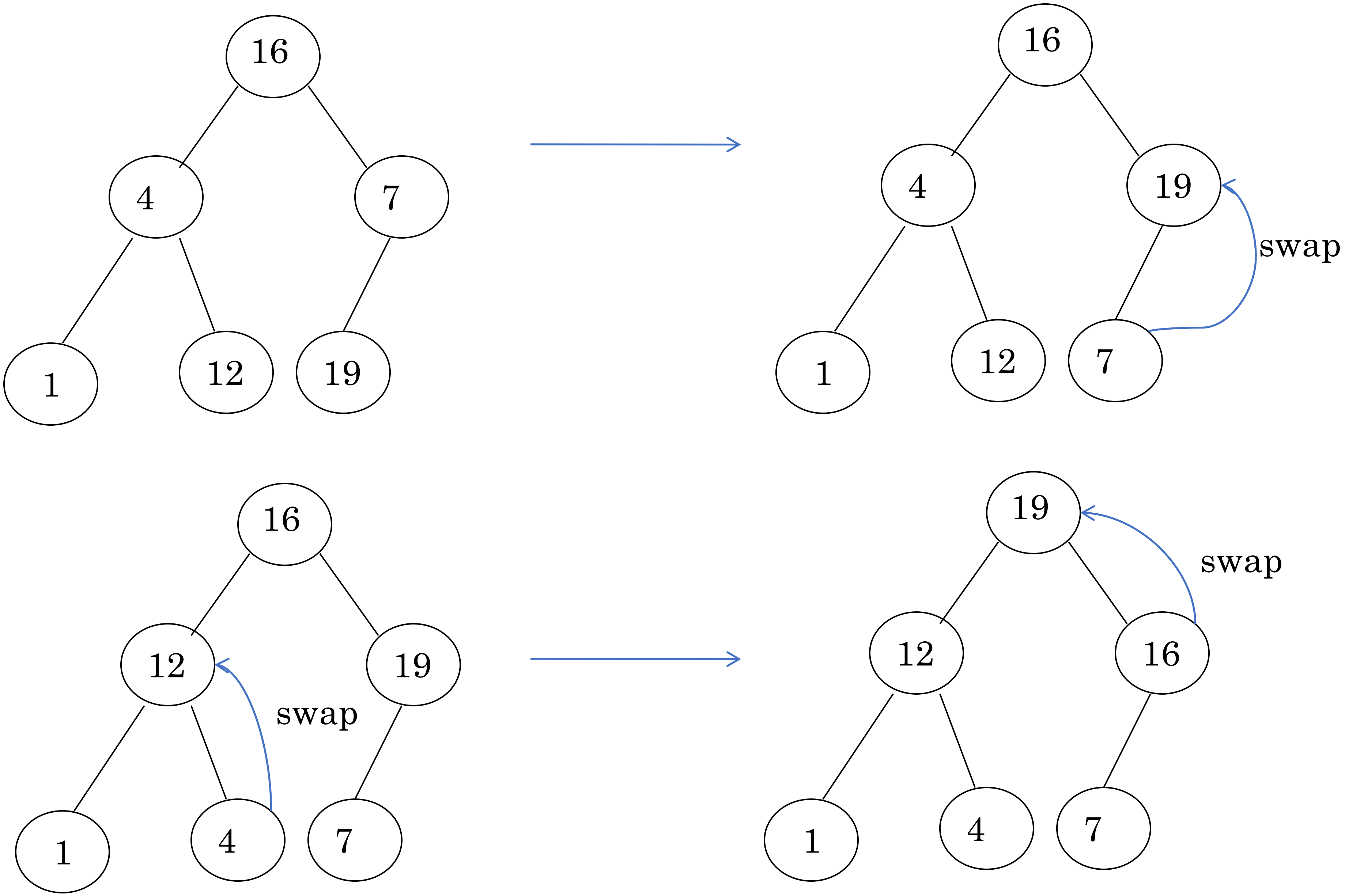
```
Heapsort(A)
{
    Buildheap(A)
    for i ← length[A]
      do swap A[1] ←→ A[i]
      heapsize[A] ← heapsize[A] - 1
      Heapify(A, 1)
}
```

**Example:** Convert the following array to a heap

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| 16 | 4 | 7 | 1 | 12 | 19 |

Picture **the array as a complete binary tree:**

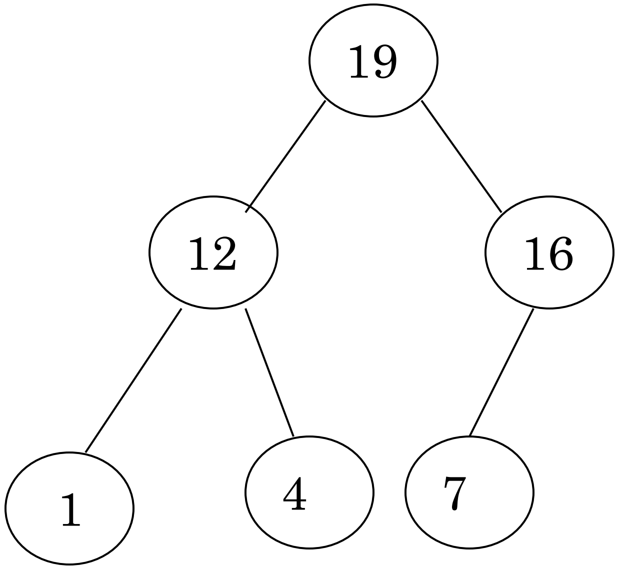



16
4
7
1
12
19
16
4
19
1
12
7
swap
16
12
19
1
4
7
swap
19
12
16
1
4
7
swap

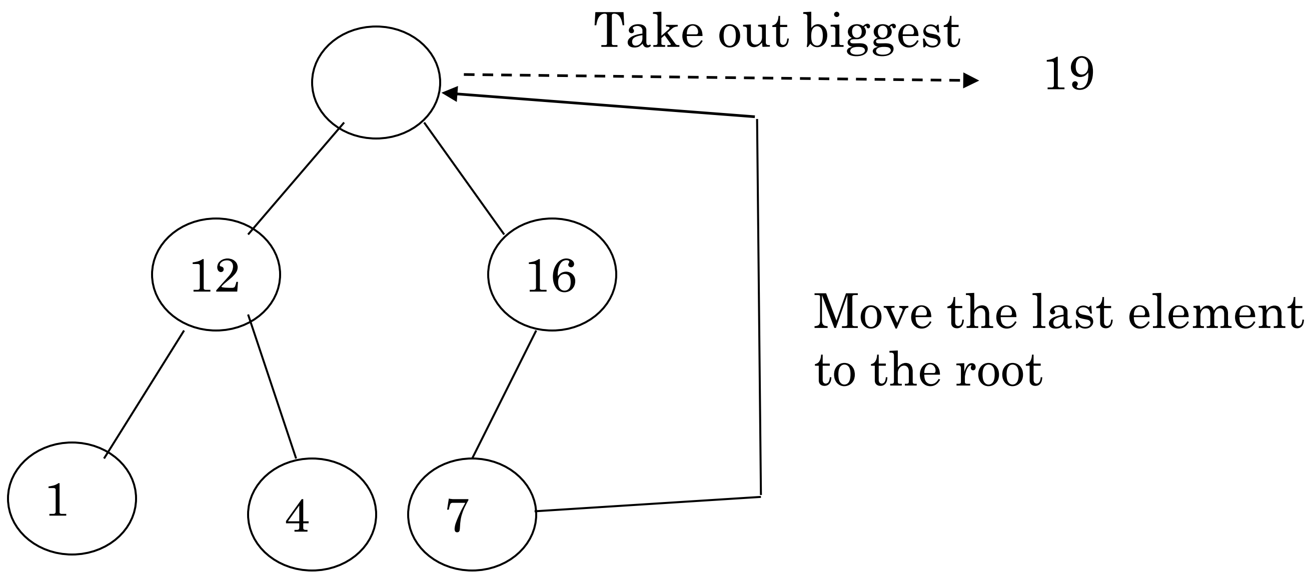
# Heap Sort

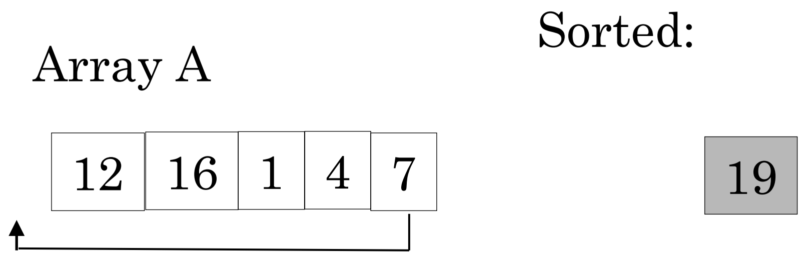
- The heapsort algorithm consists of two phases:
  - build a heap from an arbitrary array
  - use the heap to sort the data

- To sort the elements in the decreasing order, use a min heap
- To sort the elements in the increasing order, use a max heap

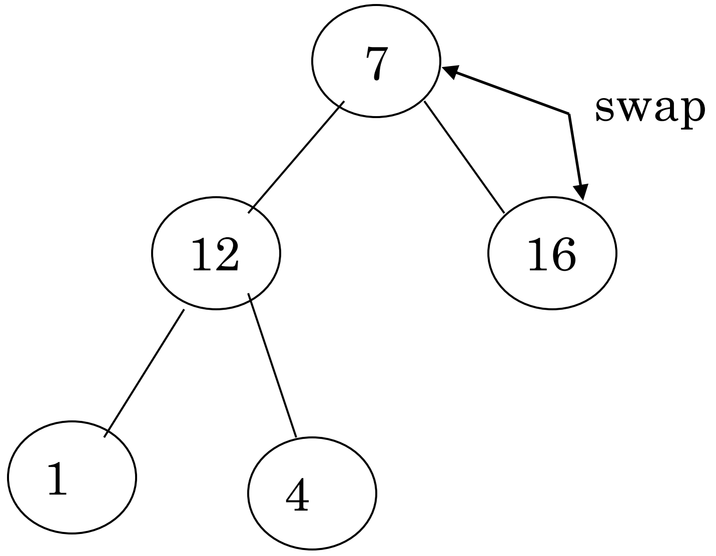


# Example of Heap Sort





HEAPIFY()

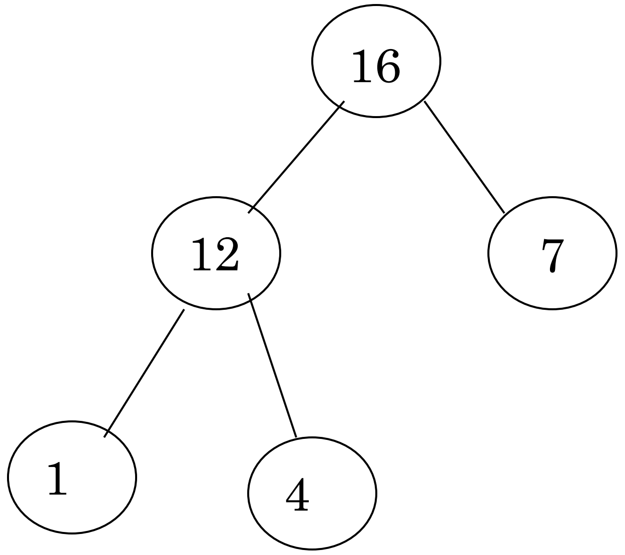


Array A

| | | | | |
|---|---|---|---|---|
| 7 | 12 | 16 | 1 | 4 |

Sorted:

| |
|---|
| 19 |

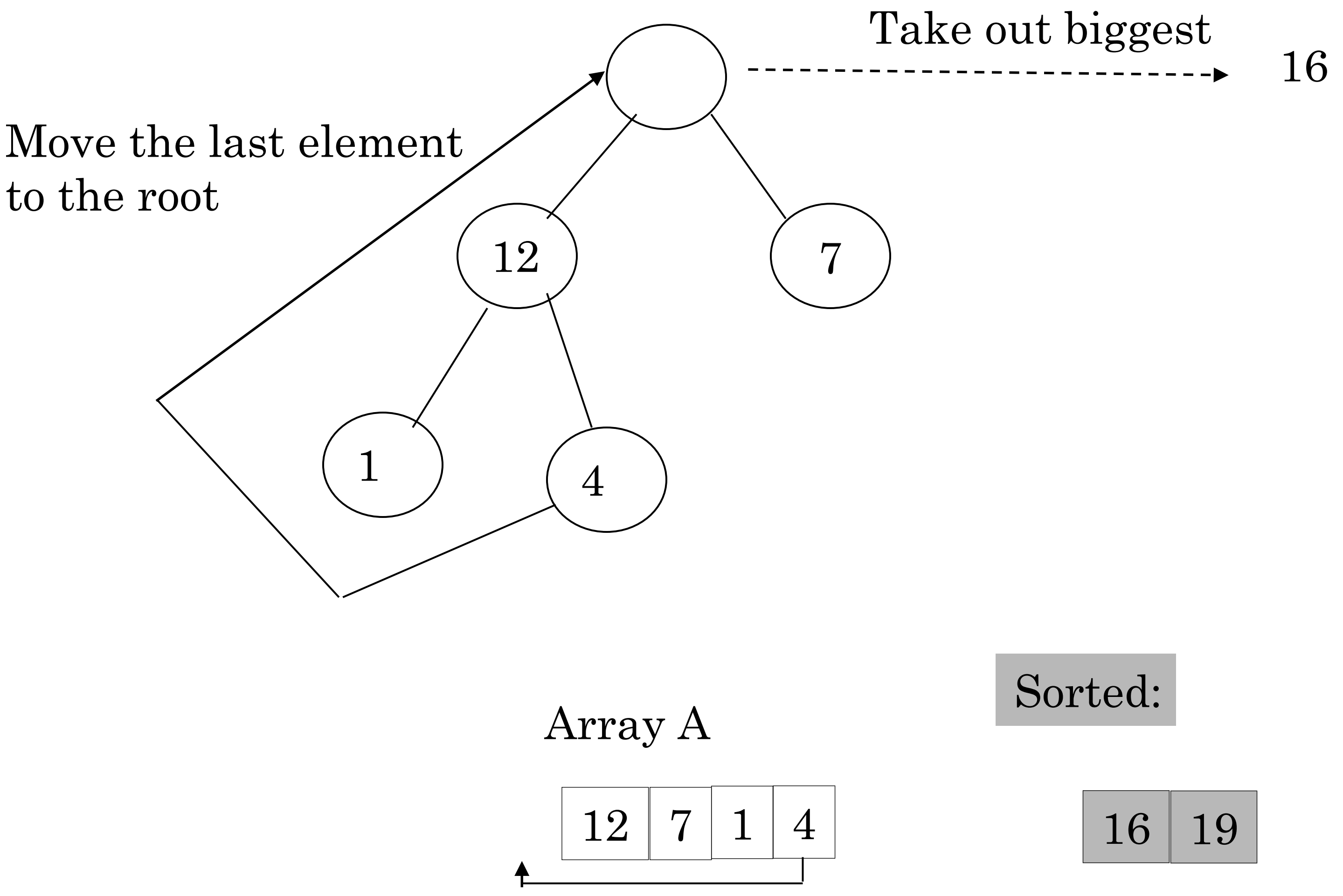


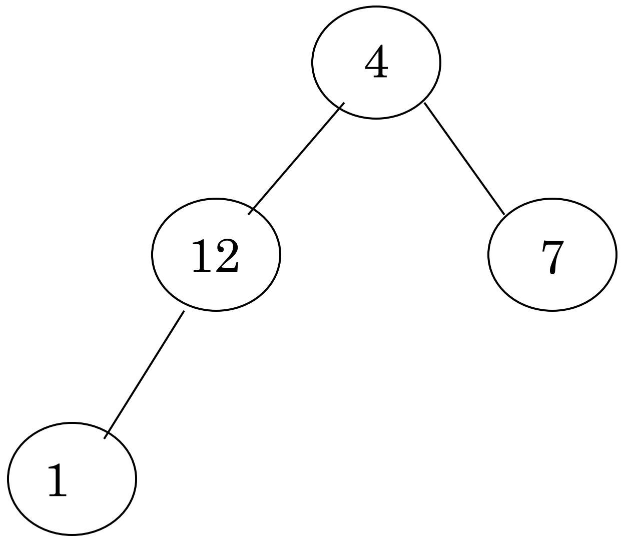
Array A

|  |  |  |  |  |
|---|---|---|---|---|
| 16 | 12 | 7 | 1 | 4 |

Sorted:

|  |
|---|
| 19 |


Take out biggest
16
Move the last element to the root
12
7
1
4
Array A
12
7
1
4
Sorted:
16
19



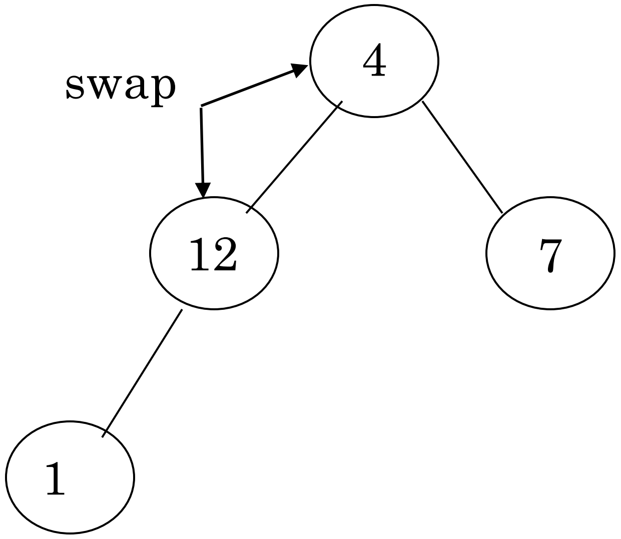
Sorted:

Array A

|  |  |  |  |
|---|---|---|---|
| 4 | 12 | 7 | 1 |

|  |  |
|---|---|
| 16 | 19 |

HEAPIFY()

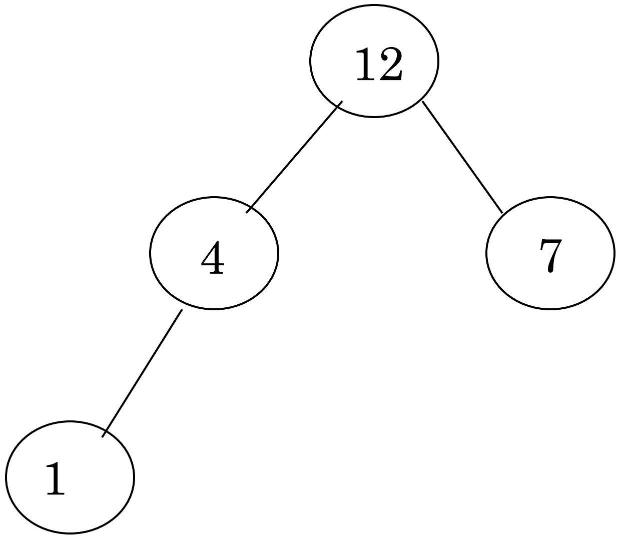


Sorted:

Array A

|  |  |  |  |
|---|---|---|---|
| 4 | 12 | 7 | 1 |

|  |  |
|---|---|
| 16 | 19 |

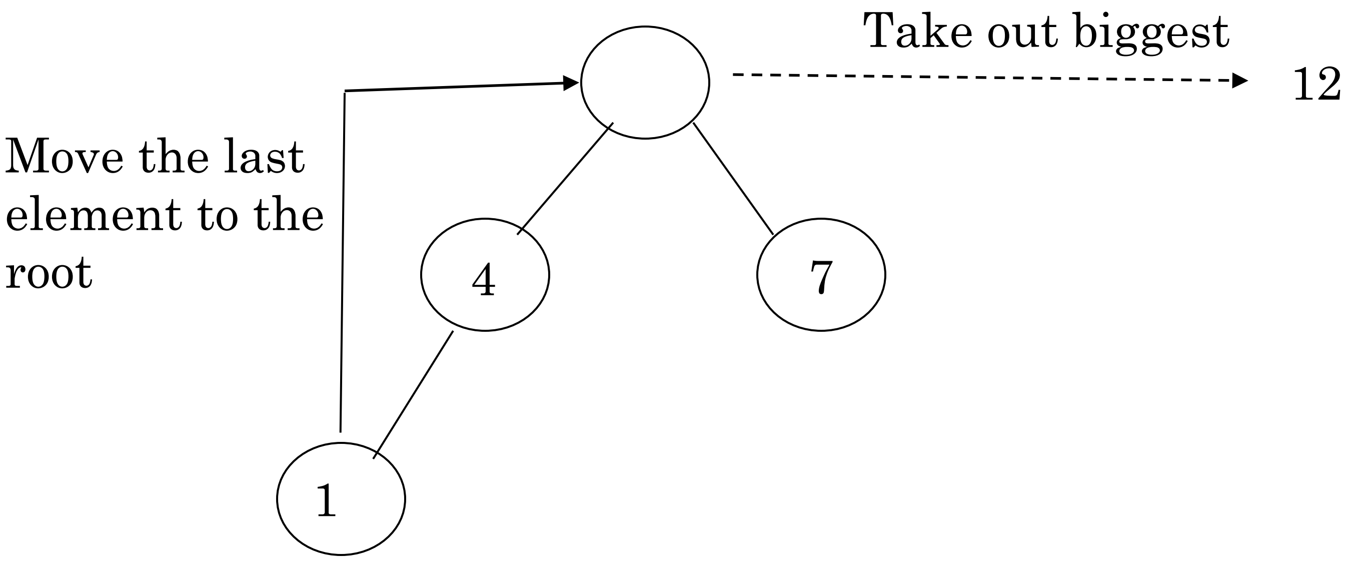


Sorted:

Array A

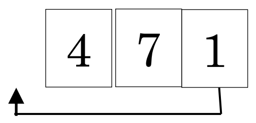
| | | | |
|---|---|---|---|
| 12 | 4 | 7 | 1 |

| | |
|---|---|
| 16 | 19 |



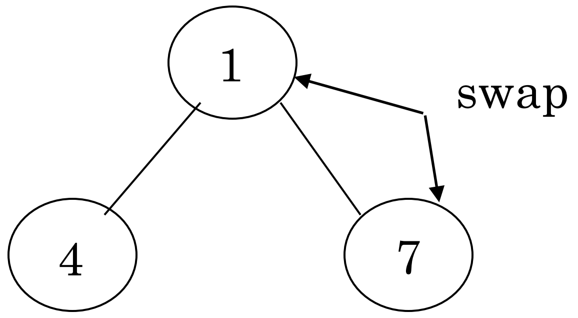
Array A

| | | |
|---|---|---|
| 4 | 7 | 1 |



Sorted:

| | | |
|---|---|---|
| 12 | 16 | 19 |

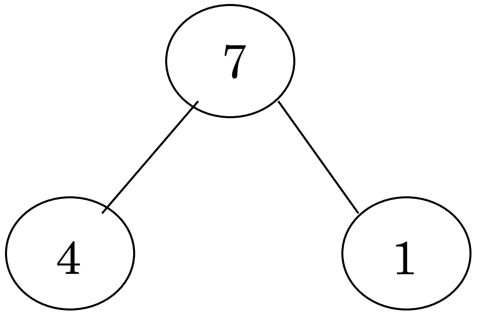


Sorted:

| | | |
|---|---|---|
| 12 | 16 | 19 |

Array A

| | | |
|---|---|---|
| 1 | 4 | 7 |

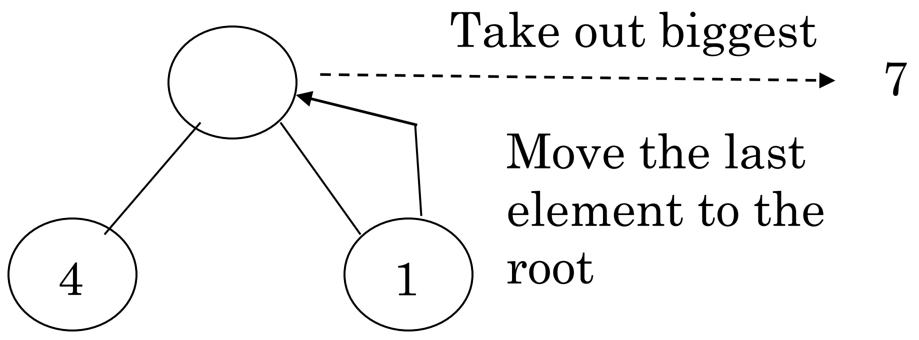


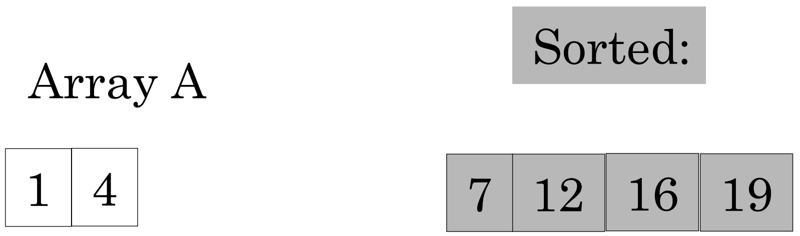
Array A

|  |  |  |
|---|---|---|
| 7 | 4 | 1 |

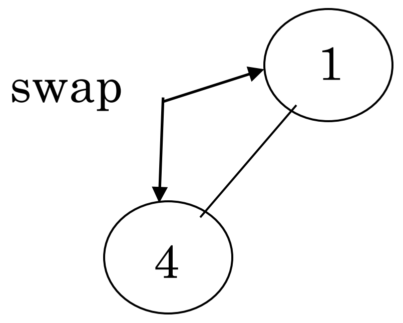
Sorted:

|  |  |  |
|---|---|---|
| 12 | 16 | 19 |


Take out biggest
7
Move the last element to the root
4
1


Array A
1
4
Sorted:
7
12
16
19

HEAPIFY()

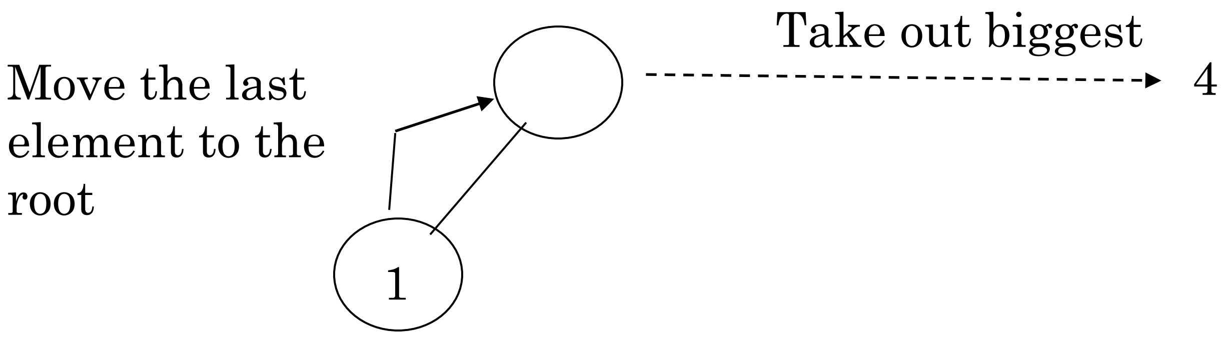


Array A

| | |
|---|---|
| 4 | 1 |

Sorted:

| | | | |
|---|---|---|---|
| 7 | 12 | 16 | 19 |

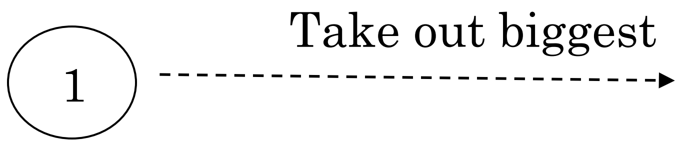


Array A

| |
|---|
| 1 |

Sorted:

| | | | | |
|---|---|---|---|---|
| 4 | 7 | 12 | 16 | 19 |



Sorted:

Array A

|   |   |   |    |    |    |
|---|---|---|----|----|----|
| 1 | 4 | 7 | 12 | 16 | 19 |

Sorted:

|   |   |   |    |    |    |
|---|---|---|----|----|----|
| 1 | 4 | 7 | 12 | 16 | 19 |

## Time Analysis

- Build Heap Algorithm will run in O(n) time
- There are *n*-1 calls to Heapify each call requires O(log *n*) time
- Heap sort program combine Build Heap program and Heapify, therefore it has the running time of O(n log n) time
- Total time complexity: O(n log n)

End of Part-II
!!!