# Chapter Eight

## Advanced Sorting Algorithms

# Revision - Sorting

- ***Sorting*** is a technique to ***rearrange the elements of a list*** in ascending or descending order.
- A ***sorting algorithm*** is an algorithm that puts elements of a list in a certain order.
- The most-used orders are ***numerical order*** and ***lexicographical order.***
- **Sorting Example:**
- Given: a set (container) of n elements
  - » E.g. array, set of words, etc.
- Suppose: there is an order relation that can be set across the elements
- Goal: Arrange the elements in ascending order
  - » Start → 1 23 2 56 9 8 10 100
  - » End → 1 2 8 9 10 23 56 100

# Sorting Algorithms

- The sorting algorithms can be categorized as:

  - » Insertion sort
  - » Bubble sort
  - » Selection sort

  **Simple Sorting Algorithms**

  - » Shell sort
  - » Quick sort
  - » Heap sort
  - » Merge sort
  - » Radix sort

  **Advanced (Efficient) Sorting Algorithms**

# Shell Sort

- The ***shell sort,*** also known as the ***diminishing increment sort***, was developed by **Donald L. Shell** in 1959.
- The idea behind shell sort is that it is faster to sort an array if parts of it are ***already sorted***.
- The original array is first divided into a number of ***smaller subarrays***, these subarrays are sorted, and then they are combined into the overall array and sorted.

# How Shell Sort Works?

- One approach would be to ***divide*** the array into a ***number of sub arrays*** consisting of contiguous elements (i.e. elements that are next to each other).
- For example, the array *[abcdef]* could be divided into the sub arrays *[abc]* and *[def]*.
- However, shell sort uses a different approach: the sub arrays are constructed by taking elements that are ***regularly spaced*** from each other.
- For example, a sub array may consist of ***every second element*** in an array, or ***every third element***, etc.
- For example, dividing the array *[abcdef]* into two sub arrays by taking every second element results in the sub arrays *[ace]* and *[bdf]*

# Cont’d

- Actually, shell sort uses several iterations of this technique.
- First, a large number of sub arrays, consisting of widely spaced elements, are sorted.
- Then, these sub arrays are combined into the overall array, a new division is made into a smaller number of sub arrays and these are sorted.
- In the next iteration a still smaller number of sub arrays is sorted.
- This process continues until eventually only one sub array is sorted, the original array itself.

| data before 5-sort | 10 | 8 | 6 | 20 | 4 | 3 | 22 | 1 | 0 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 subarrays before 5-sort | 10 | | | | | 3 | | | | | 16 |
| | | 8 | | | | | 22 | | | | |
| | | | 6 | | | | | 1 | | | |
| | | | | 20 | | | | | 0 | | |
| | | | | | 4 | | | | | 15 | |
| 5 subarrays after 5-sort | 3 | | | | | 10 | | | | | 16 |
| | | 8 | | | | | 22 | | | | |
| | | | 1 | | | | | 6 | | | |
| | | | | 0 | | | | | 20 | | |
| | | | | | 4 | | | | | 15 | |
| data after 5-sort and before 3-sort | 3 | 8 | 1 | 0 | 4 | 10 | 22 | 6 | 20 | 15 | 16 |
| 3 subarrays before 3-sort | 3 | | | 0 | | | 22 | | | 15 | |
| | | 8 | | | 4 | | | 6 | | | 16 |
| | | | 1 | | | 10 | | | 20 | | |
| 3 subarrays after 3-sort | 0 | | | 3 | | | 15 | | | 22 | |
| | | 4 | | | 6 | | | 8 | | | 16 |
| | | | 1 | | | 10 | | | 20 | | |
| data after 3-sort and before 1-sort | 0 | 4 | 1 | 3 | 6 | 10 | 15 | 8 | 20 | 22 | 16 |
| data after 1-sort | 0 | 1 | 3 | 4 | 6 | 8 | 10 | 15 | 16 | 20 | 22 |

# Cont'd

- In the above example, we used ***three iterations***: ***a 5-sort, a 3-sort and a 1-sort.***
- This sequence is known as the ***diminishing increment***.
- But how do we decide on this sequence of increments?
  - Powers of 2 were used for the increments,
  - e.g. 16, 8, 4, 2, 1.
- However, this is not the most efficient technique.
- Experimental studies have shown that increments calculated according to the following conditions lead to better efficiency:

  $h_1 = 1$

  $h_{i+1} = 3h_i + 1$
- For example, for a list of length 100 the sequence of increments would be 40, 13, 4, 1.

# Cont'd

- An experimental analysis has shown that the complexity of ***shell sort*** is approximately ***O(n^1.25^),***
  - » which is better than the ***O(n²)*** offered by the simple algorithms.
- But it is advisable to choose $g_k$ =n/2 and $g_{k-1}$ = $g_i$/2 for k>=i>=1. After each sequence $g_{k-1}$ is done and the list is said to be $g_i$-sorted.
- Shell sorting is done when the list is *1-sorted* (which is sorted using insertion sort) and A[j]<=A[j+1] for 0<=j<=n-2. Time complexity is ***$O(n^{3/2})$.***

# Cont'd

- **Example:** Sort the following list using shell sort algorithm.
  - » 5 8 2 4 1 3 9 7 6 0
- Choose $g_3$ =5 (n/2 where n is the number of elements =10)
- Sort (5, 3)
  - » 3 8 2 4 1 5 9 7 6 0
- Sort (8, 9)
  - » 3 8 2 4 1 5 9 7 6 0
- Sort (2, 7)
  - » 3 8 2 4 1 5 9 7 6 0
- Sort (4, 6)
  - » 3 8 2 4 1 5 9 7 6 0

# Cont'd

- Sort (1, 0)
  - 3 8 2 4 0 5 9 7 6 1
- 5- sorted list
  - 3 8 2 4 0 5 9 7 6 1
- Sort (3, 4, 9, 1)
  - 1 8 2 3 0 5 4 7 6 9
- Sort (8, 0, 7)
  - 1 0 2 3 7 5 4 8 6 9
- Sort (2, 5, 6)
  - 1 0 2 3 7 5 4 8 6 9
- 3- sorted list
  - 1 0 2 3 7 5 4 8 6 9

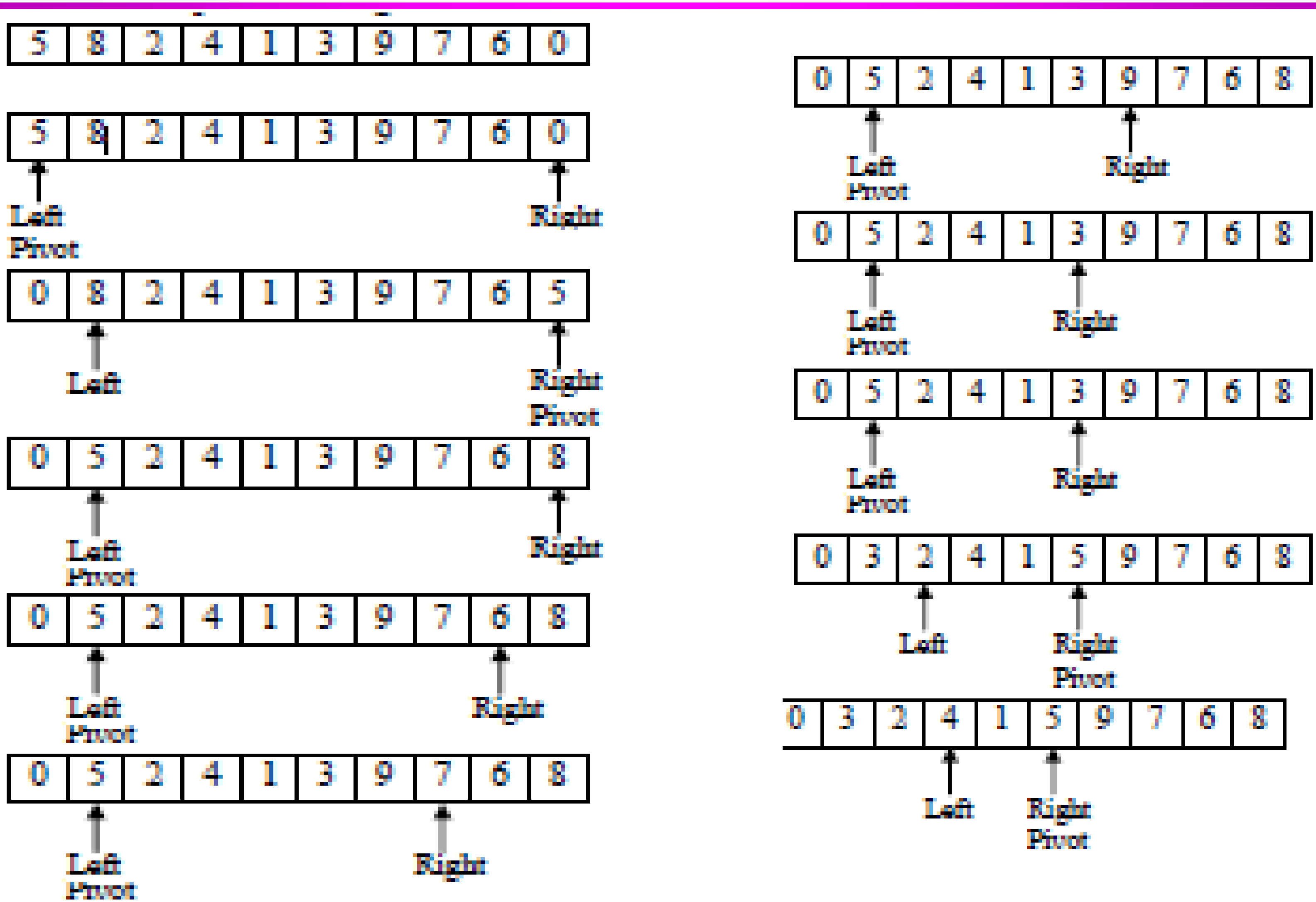
# Cont'd

- Choose g*1* =1 (the same as insertion sort algorithm)
- Sort (1, 0, 2, 3, 7, 5, 4, 8, 6, 9)
  - » 0 1 2 3 4 5 6 7 8 9
- 1- sorted (shell sorted) list
  - » 0 1 2 3 4 5 6 7 8 9

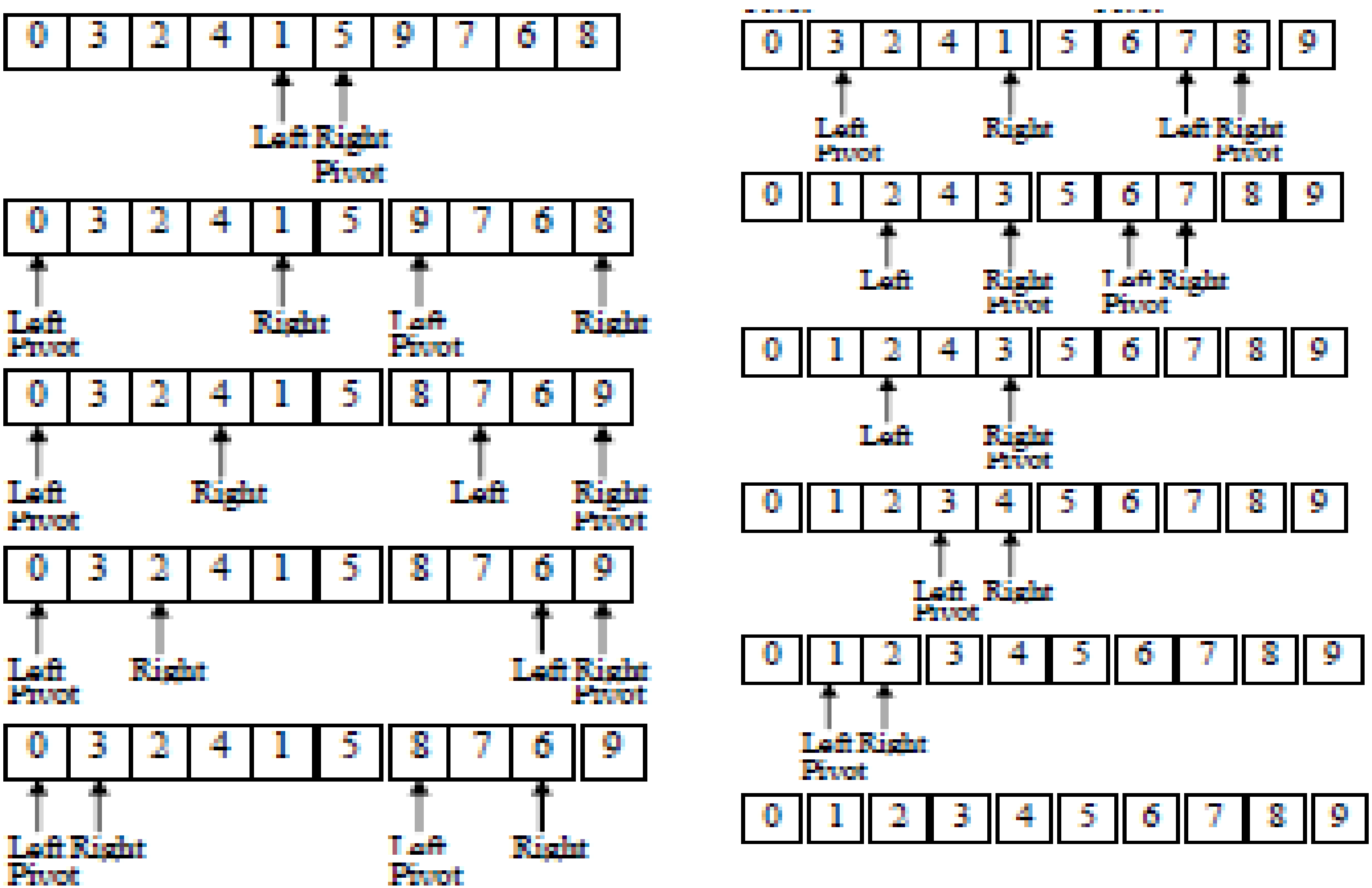
# Quick Sort

- Quick sort is the ***fastest*** known algorithm.
- It uses ***divide and conquer*** strategy and in the worst case its complexity is O ($n^2$).
- But its expected complexity is O(nlogn).
- **Algorithm:**
  - Choose a ***pivot value*** (mostly the ***first element*** is taken as the pivot value)
  - Position the pivot element and partition the list so that:
    - the ***left part*** has items less than or equal to the pivot value
    - the ***right part*** has items greater than or equal to the pivot value
  - Recursively ***sort*** the ***left*** part
  - Recursively ***sort*** the ***right*** part

# Example


5 8 2 4 1 3 9 7 6 0
5 8 2 4 1 3 9 7 6 0
Left
Pivot
Right
0 8 2 4 1 3 9 7 6 5
Left
Right
Pivot
0 5 2 4 1 3 9 7 6 8
Left
Pivot
Right
0 5 2 4 1 3 9 7 6 8
Left
Pivot
Right
0 5 2 4 1 3 9 7 6 8
Left
Pivot
Right
0 5 2 4 1 3 9 7 6 8
Left
Pivot
Right
0 5 2 4 1 3 9 7 6 8
Left
Pivot
Right
0 5 2 4 1 3 9 7 6 8
Left
Pivot
Right
0 3 2 4 1 5 9 7 6 8
Left
Right
Pivot
0 3 2 4 1 5 9 7 6 8
Left
Right
Pivot

# Cont’d



0 3 2 4 1 5 9 7 6 8
Left Right
Pivot
0 3 2 4 1 5 9 7 6 8
Left
Pivot
Right
Left
Pivot
Right
0 3 2 4 1 5 8 7 6 9
Left
Pivot
Right
Left
Right
Pivot
0 3 2 4 1 5 8 7 6 9
Left
Pivot
Right
Left Right
Pivot
0 3 2 4 1 5 8 7 6 9
Left Right
Pivot
Left
Pivot
Right
0 3 2 4 1 5 6 7 8 9
Left
Pivot
Right
Left Right
Pivot
0 1 2 4 3 5 6 7 8 9
Left
Right
Pivot
Left Right
Pivot
0 1 2 4 3 5 6 7 8 9
Left
Right
Pivot
0 1 2 3 4 5 6 7 8 9
Left
Pivot
Right
0 1 2 3 4 5 6 7 8 9
Left Right
Pivot
0 1 2 3 4 5 6 7 8 9

# Implementation

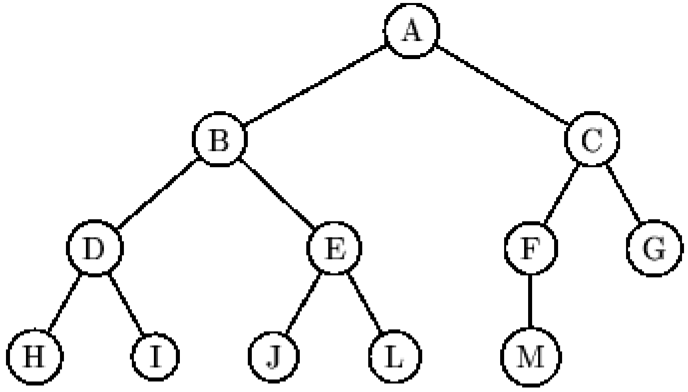
```
Left=0;
Right=n-1; // n is the total number of elements in the list
PivotPos=Left;
while(Left<Right)
{
    if(PivotPos==Left)
    {
          if(Data[Left]>Data[Right])
          {
          swap(data[Left], Data[Right]);
          PivotPos=Right;
          Left++;
          }
          else
          Right--;
    }
```

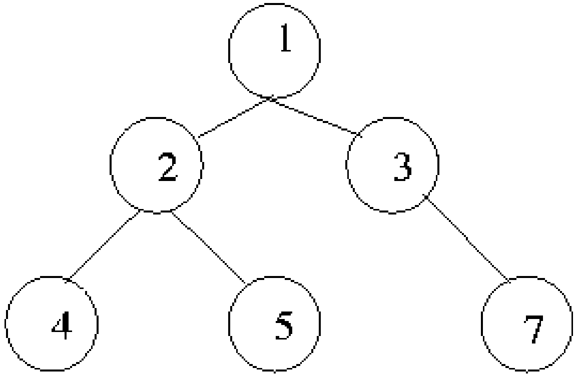
# Cont'd

```
else
{
    if(Data[Left]>Data[Right])
    {
    swap(data[Left], Data[Right]);
    PivotPos=Left;
    Right--;
    }
    else
    Left++;
}
}
```

# Heap Sort

- A ***heap*** is a ***complete binary tree*** with the property that the value at each node is ***at least as large as*** the values at its ***children***.
- A ***largest element*** is at the ***root*** of the heap.
- ***Complete binary tree*** is a binary tree in which the level of the any leaf node is either H or H-1 where H is the height of the tree. The deepest level should also be filled from left to right.




Not Complete

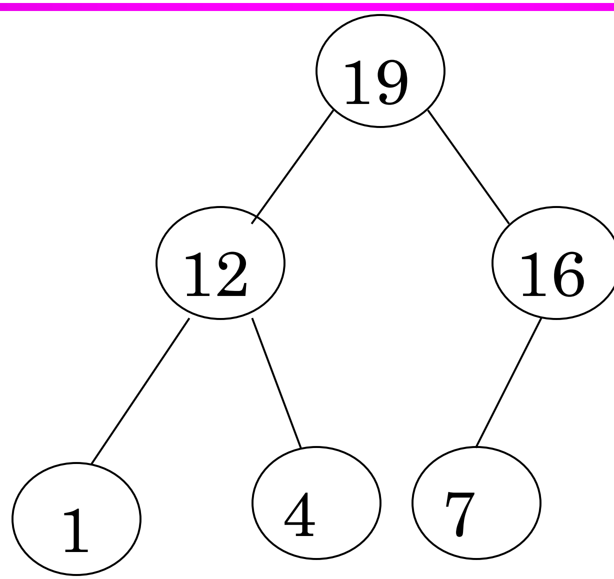
# Cont'd

- The relation ***greater than or equal to*** may be ***reversed*** so that the parent node contains a value ***as small as or smaller than its children.***
- Max Heap
  - » Store data in ascending order
  - » Has property of

    A[Parent(i)] ≥ A[i]
- Min Heap
  - » Store data in descending order
  - » Has property of

    A[Parent(i)] ≤ A[i]

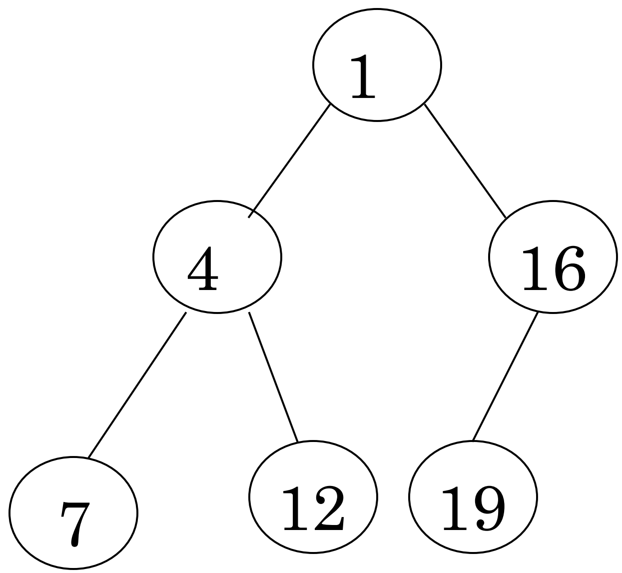
# Max Heap Example




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

Array A

# Min Heap Example



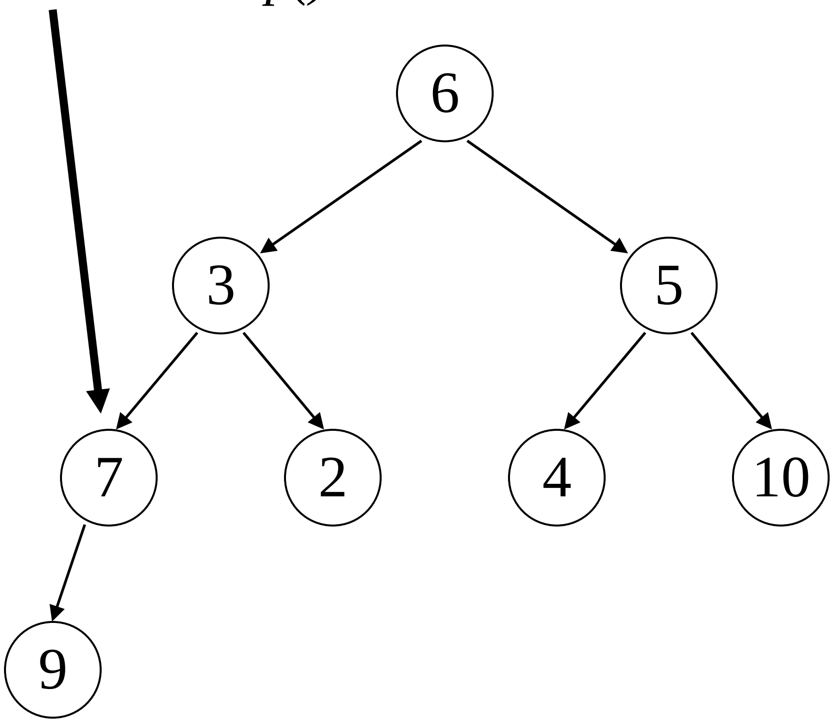

| | | | | | |
|---|---|---|---|---|---|
| 1 | 4 | 16 | 7 | 12 | 19 |

Array A

# Transform an Array Into a Heap: *Example*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 3 | 5 | 7 | 2 | 4 | 10 | 9 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

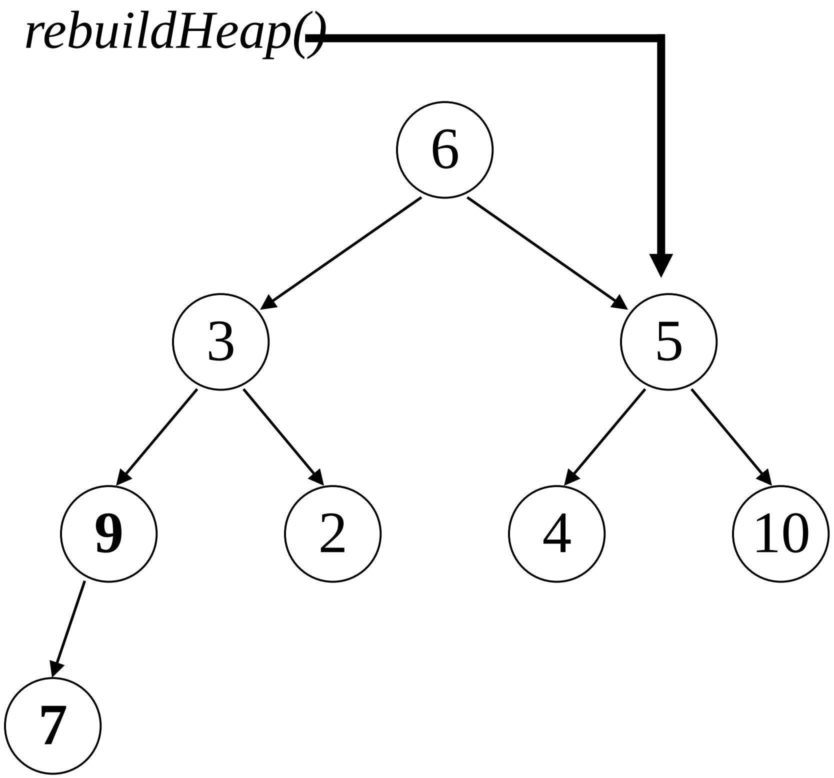
*rebuildHeap()*




- The items in the array, above, can be considered to be stored in the ***complete binary tree*** shown at right.
- *rebuildHeap()* is invoked on the *parent* of the last node in the array (= 9).

# Cont'd

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 3 | 5 | **9** | 2 | 4 | 10 | **7** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

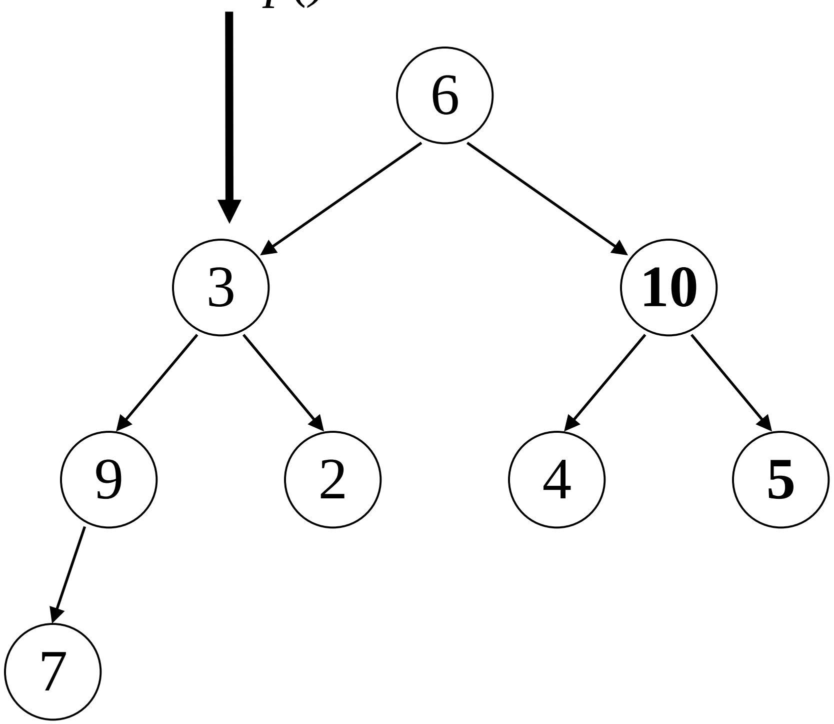



- *rebuildHeap()* is invoked on the node in the array *preceding* node 9.

# Cont'd

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 3 | **10** | 9 | 2 | 4 | **5** | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

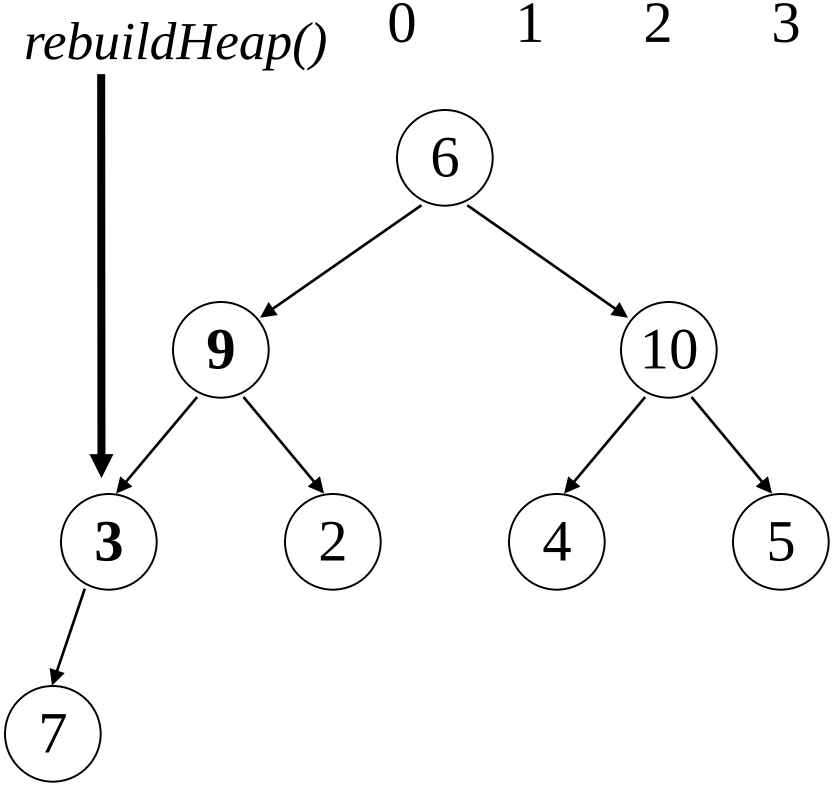
*rebuildHeap()*




- *rebuildHeap()* is invoked on the node in the array *preceding* node 10.

# Cont’d

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | **9** | 10 | **3** | 2 | 4 | 5 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

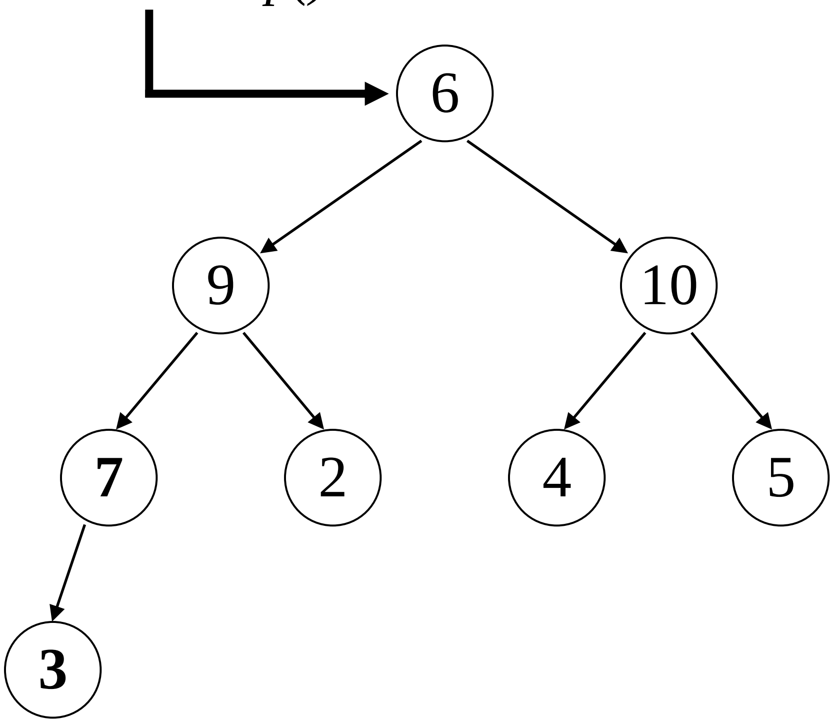



- *rebuildHeap()* is invoked recursively on node 3 to complete the transformation of the *semiheap* rooted at 9 into a *heap*.

# Cont'd

| 6 | 9 | 10 | **7** | 2 | 4 | 5 | **3** |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

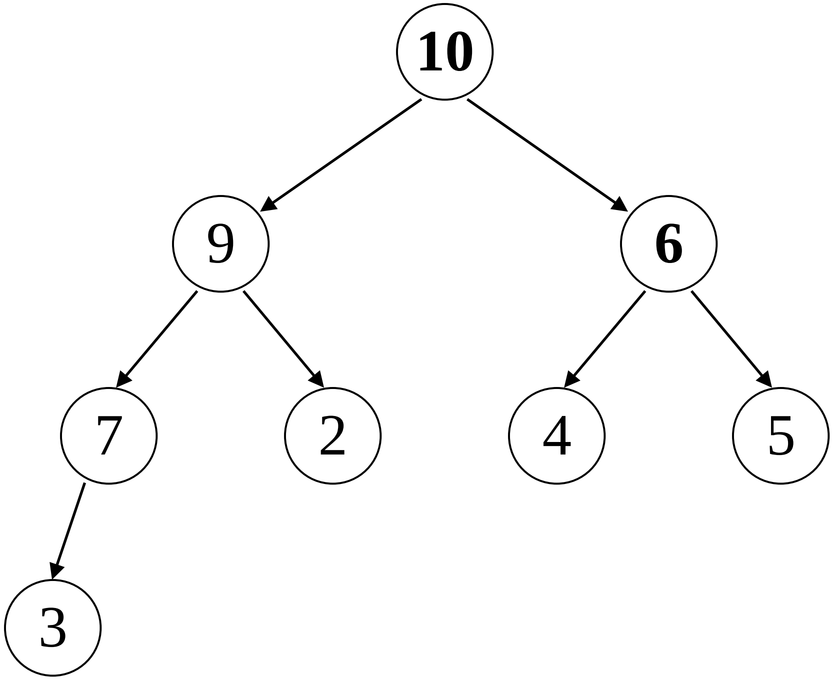
*rebuildHeap()*




- The recursive call to *rebuildHeap()* returns to node 9.
- *rebuildHeap()* is invoked on the node in the array *preceding* node 9.

# Cont'd

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **10** | 9 | **6** | 7 | 2 | 4 | 5 | 3 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |




- Note that node 10 is now the root of a *heap*.
- The transformation of the *array* into a *heap* is complete.

# Exercise

- Form a heap from the set 40,80,35,90,45,50,70.

# 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 then 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 ←|length[A]/2  //down to 1
        do Heapify(A, i)
}
```

# Heap Sort Algorithm

- The heap sort algorithm starts by using procedure BUILD-HEAP to build a heap on the input array A[1 . . *n*].
- Since the maximum element of the array stored at the root *A*[1], it can be put into its correct final position by exchanging it with *A*[*n*] (the last element in *A*).
- If we now discard node n from the heap then the remaining elements can be made into heap.
- Note that the new element at the root may violate the heap property. All that is needed to restore the heap property.

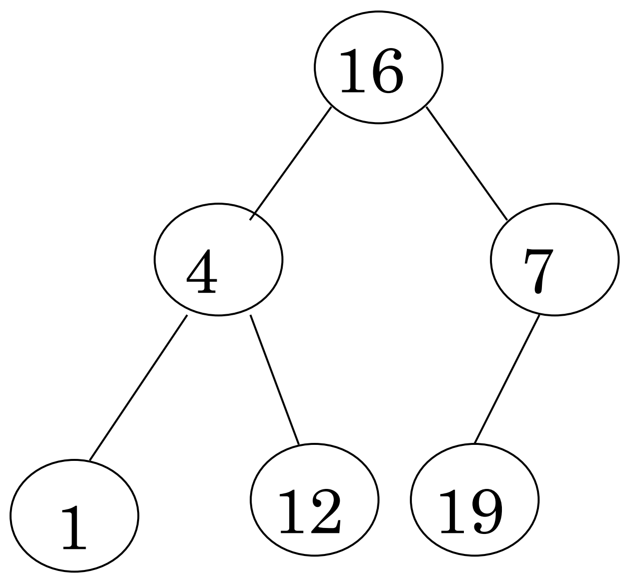
```
Heapsort(A)
{       Buildheap(A)
      for i ← length[A] //down to 2
         do swap A[1] ←→ A[i]
         heapsize[A] ← heapsize[A] - 1
         Heapify(A, 1)
}
```

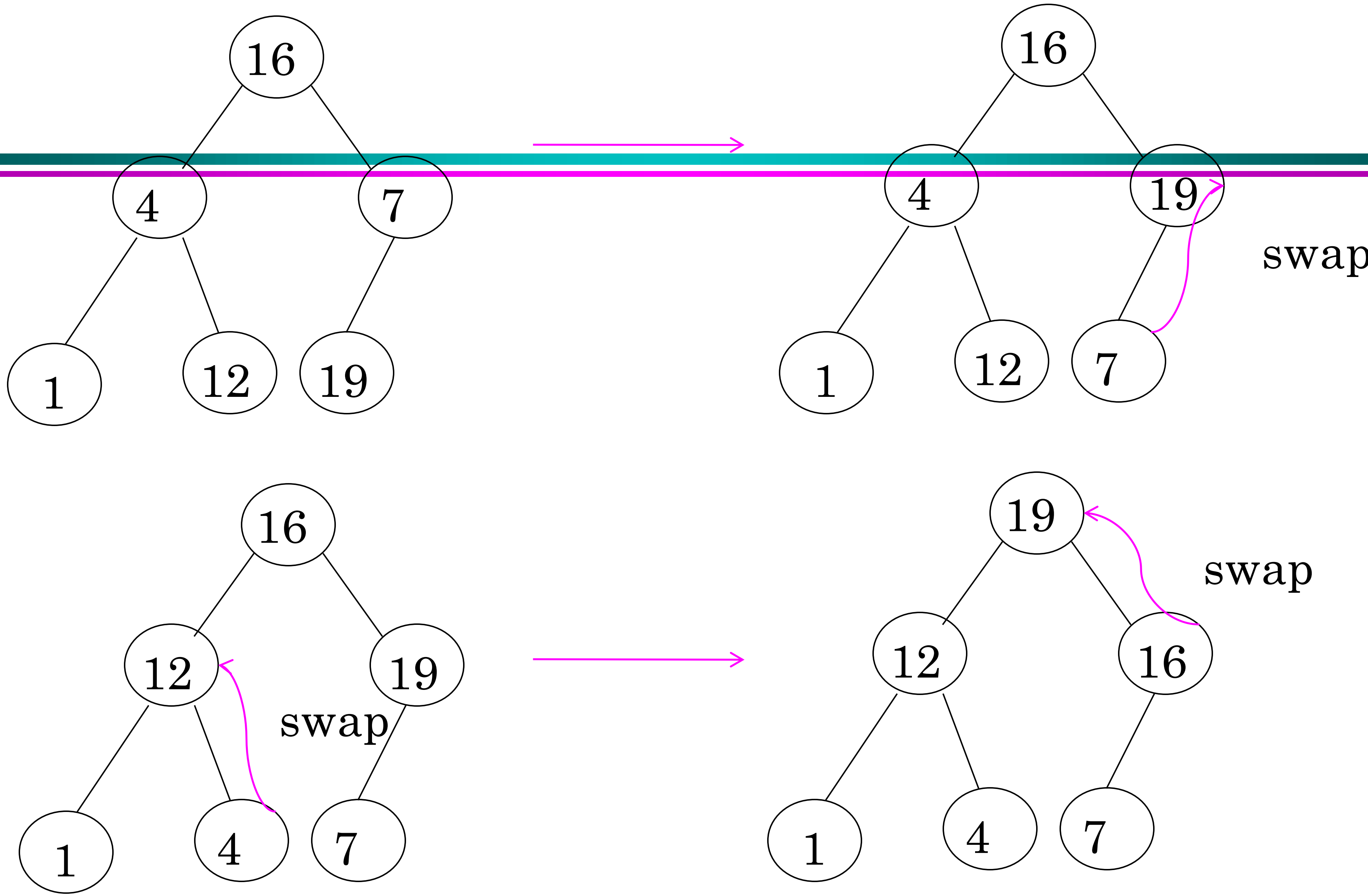
# Cont'd

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

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

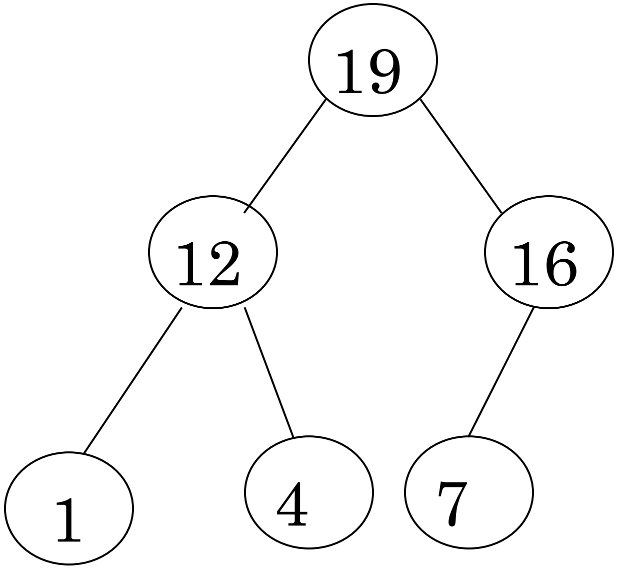
Picture **the array as a complete binary tree:**



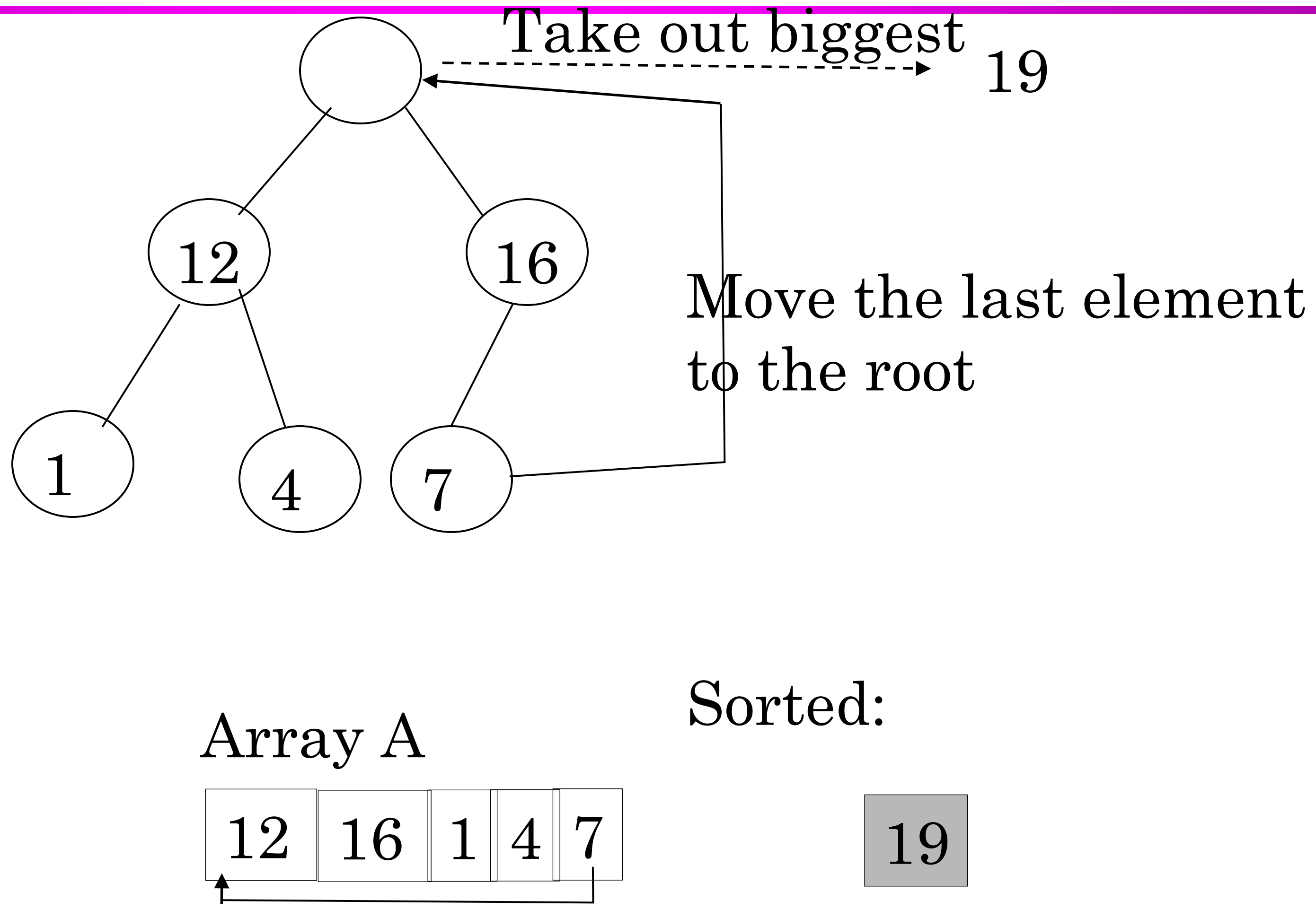

16
4
7
1
12
19
16
4
19
swap
1
12
7
16
12
19
swap
1
4
7
19
swap
12
16
1
4
7

# Cont'd

- 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

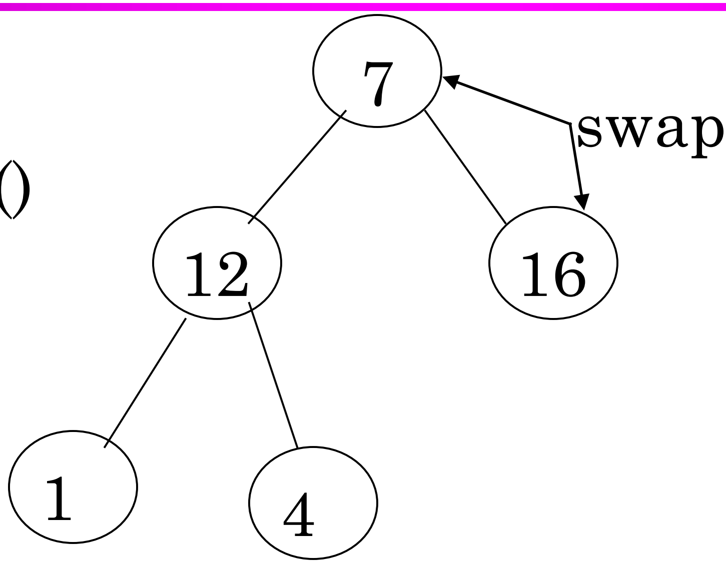


# Cont'd



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

HEAPIFY()

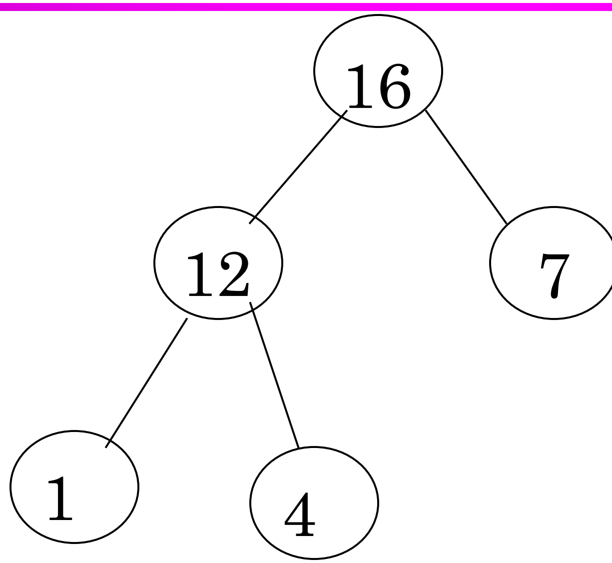



Array A

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

Sorted:

19

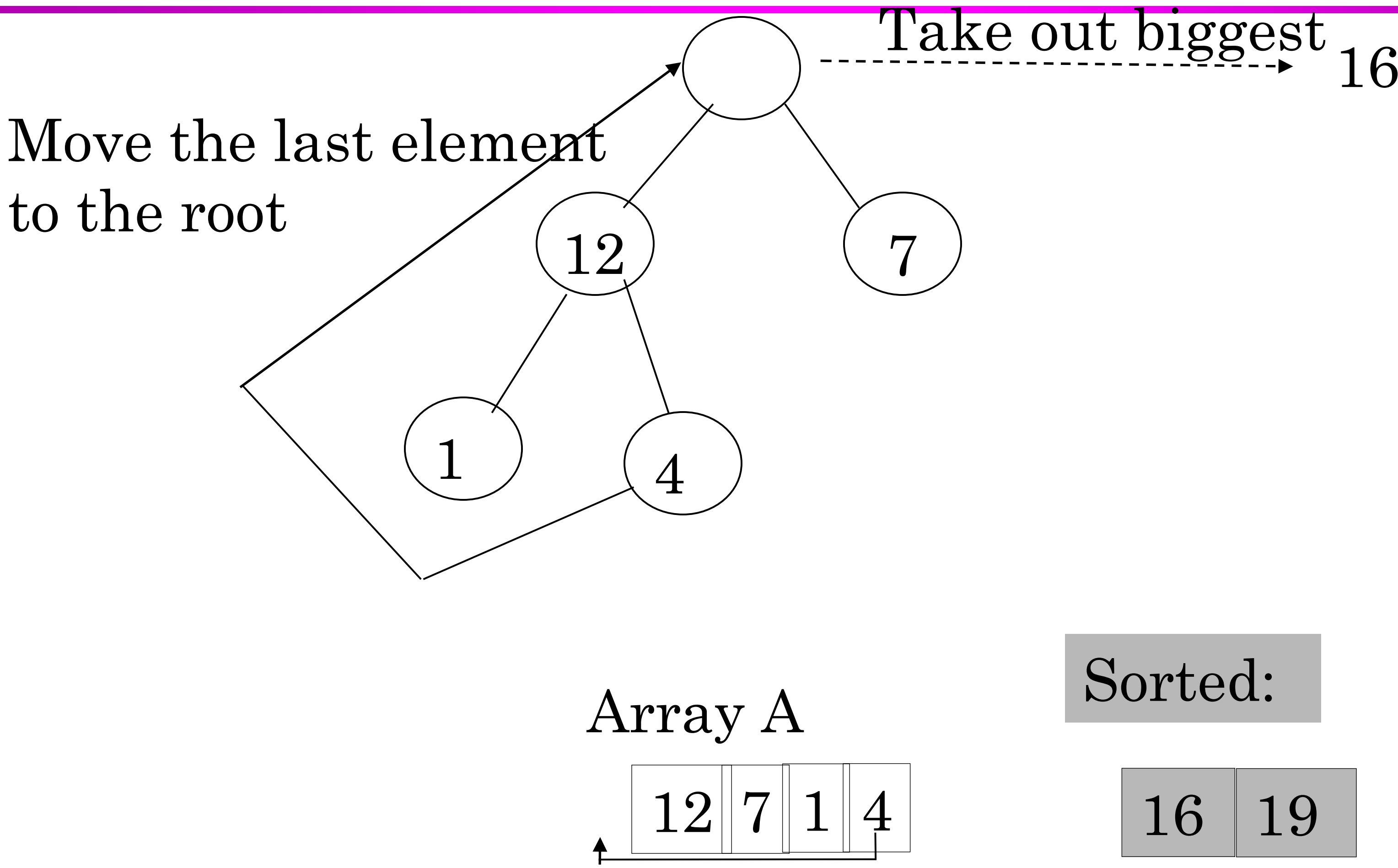


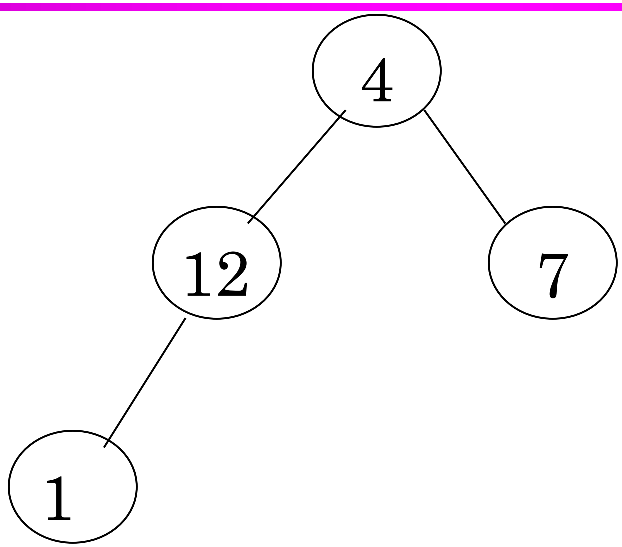

Sorted:

19

Array A

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


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



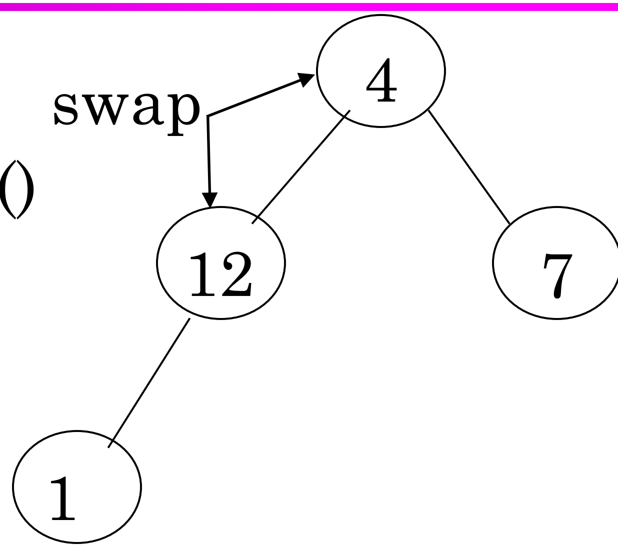

Sorted:

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

Array A

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

HEAPIFY()

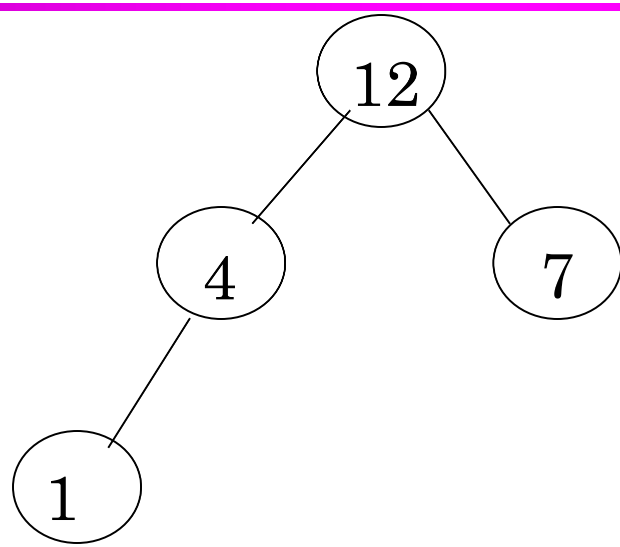



Array A

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

Sorted:

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



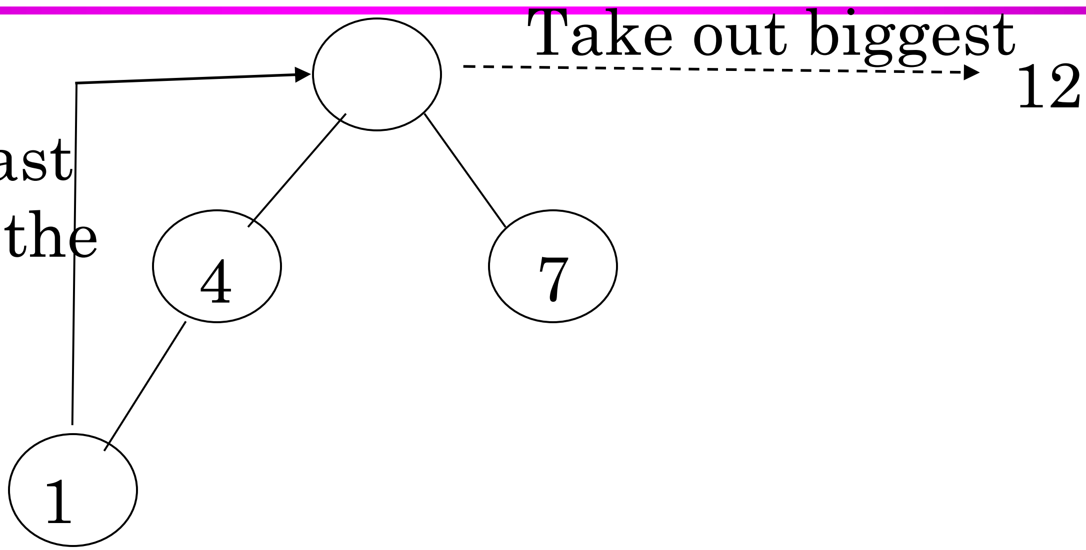

Array A

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

Sorted:

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

Move the last element to the root

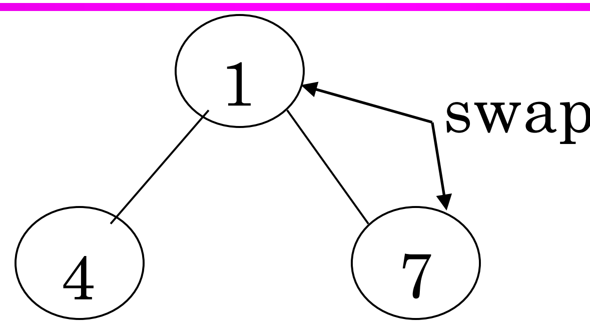



Array A

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

Sorted:

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

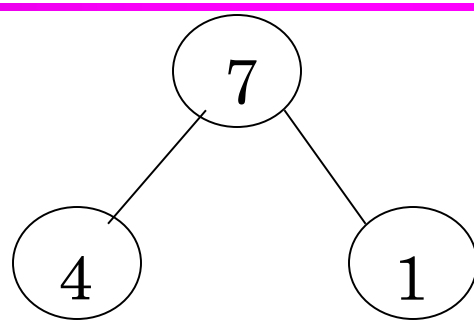



Array A

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

Sorted:

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

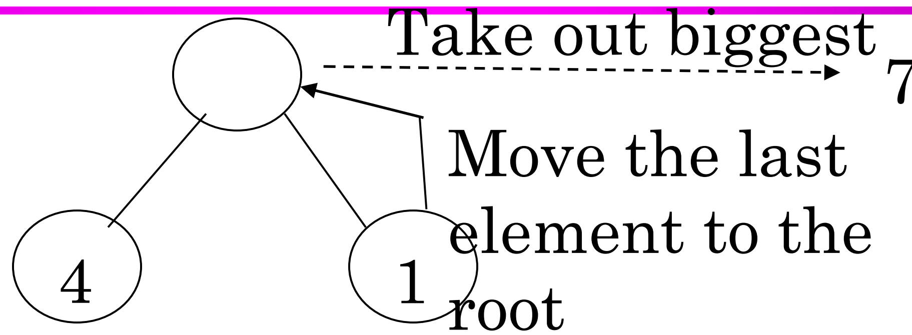



Array A

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

Sorted:

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



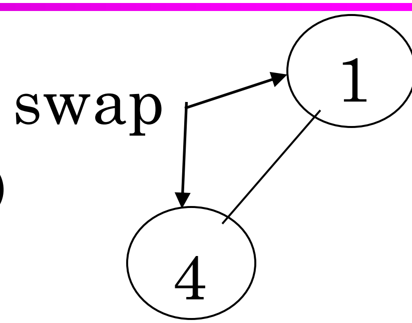

Array A

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

Sorted:

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

HEAPIFY()



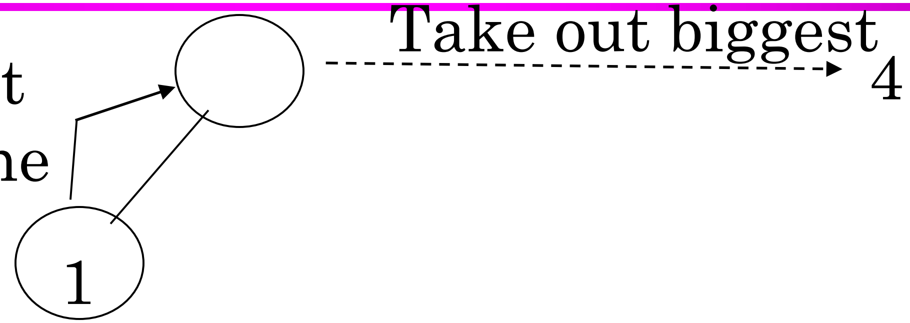

Array A

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

Sorted:

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

Move the last element to the root




Array A

| |
|---|
| 1 |

Sorted:

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

1

Take out biggest

Array A

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(nlogn) time.
- Total time complexity: O(nlogn).

- ***Reference for Visualization:***
  - https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

# Reading Assignment

- Merge Sort and
- Radix Sort