Heap Sort

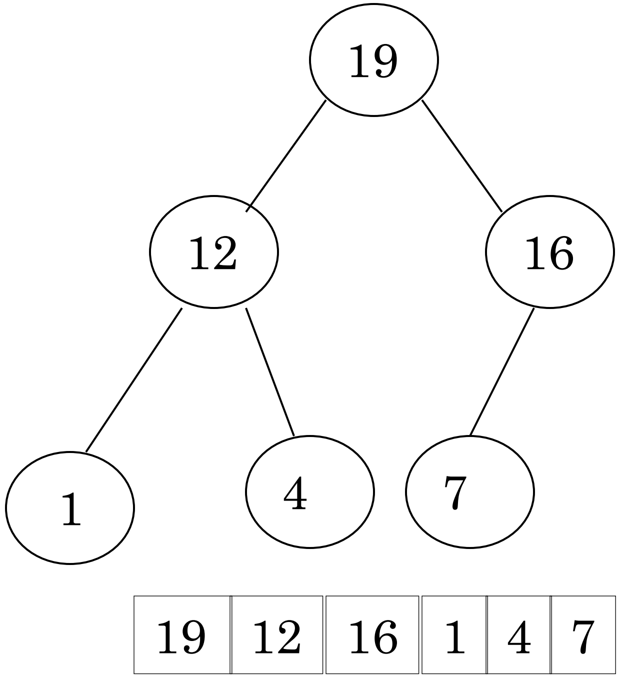
Heap

- A heap is a data structure that stores a collection of objects (with keys), and has the following properties:
    1. Complete Binary tree
    2. 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 2i
RIGHT (i)
      return 2i + 1
```

Heap order property

- For every node *v*, other than the root, the key stored in *v* is greater or equal (smaller or equal for max heap) than the key stored in the parent of *v*.

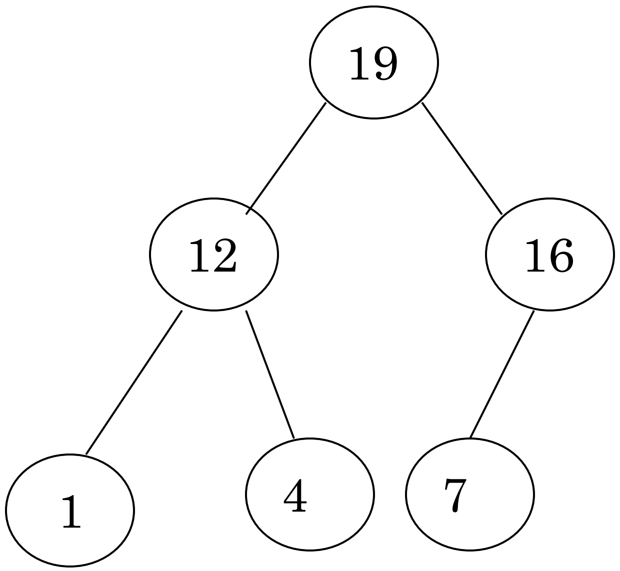
- In this case the maximum value is stored in the root

Definition

- Max Heap
  - Has property of:
    A[Parent(i)] ≥ A[i]

- Min Heap
  - Has property of:
    A[Parent(i)] ≤ A[i]

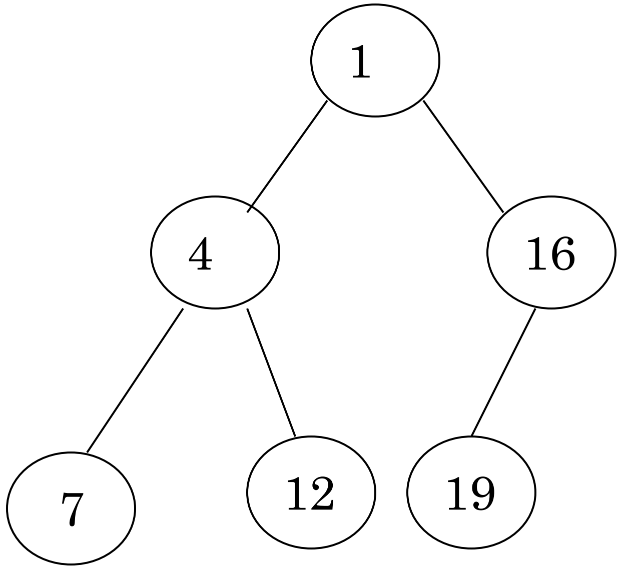
Max Heap Example



19	12	16	1	4	7

Array A

Min heap example



1	4	16	7	12	19

Array A

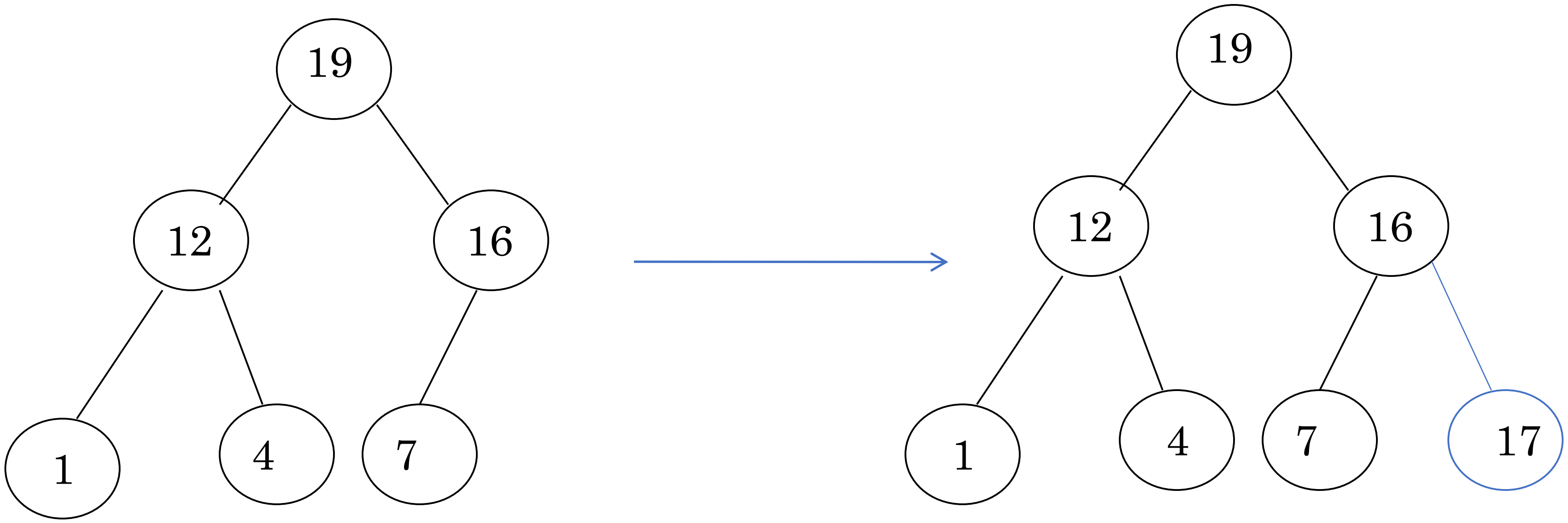
Insertion

- 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.

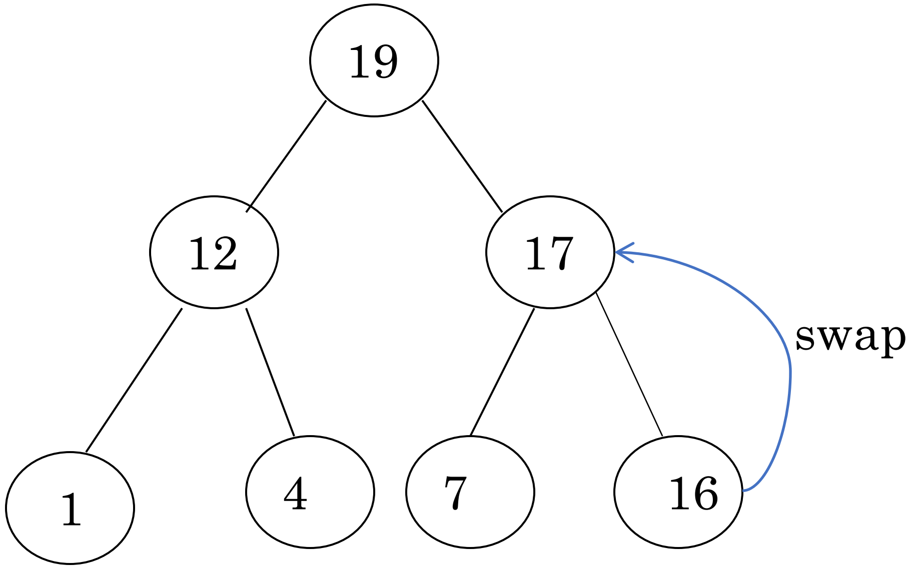
OR

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 compares 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 becomes larger than its children.

Heapify

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

- The elements in the subarray A[*n*/2 +1 . . *n*] are all leaves (so each of these elements is a 1-element heap to begin with).

- 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.

Build Heap

```
Buildheap(A)
{
    A.heapsize ← A.length
    for i ← floor(A.length/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 than 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.

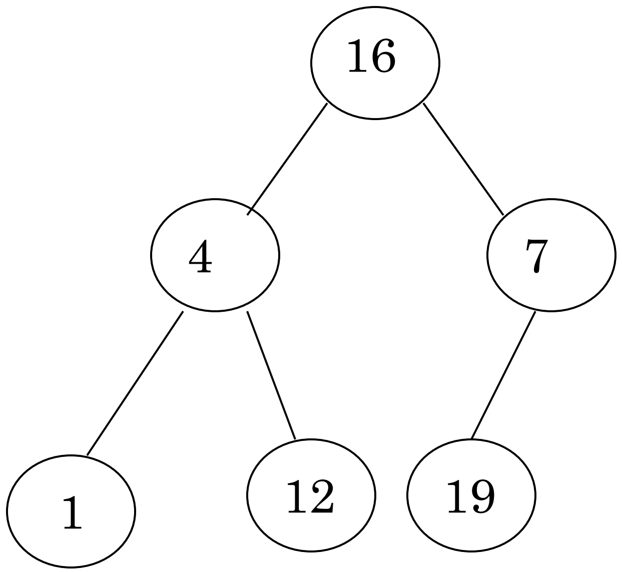
Heap Sort Algorithm

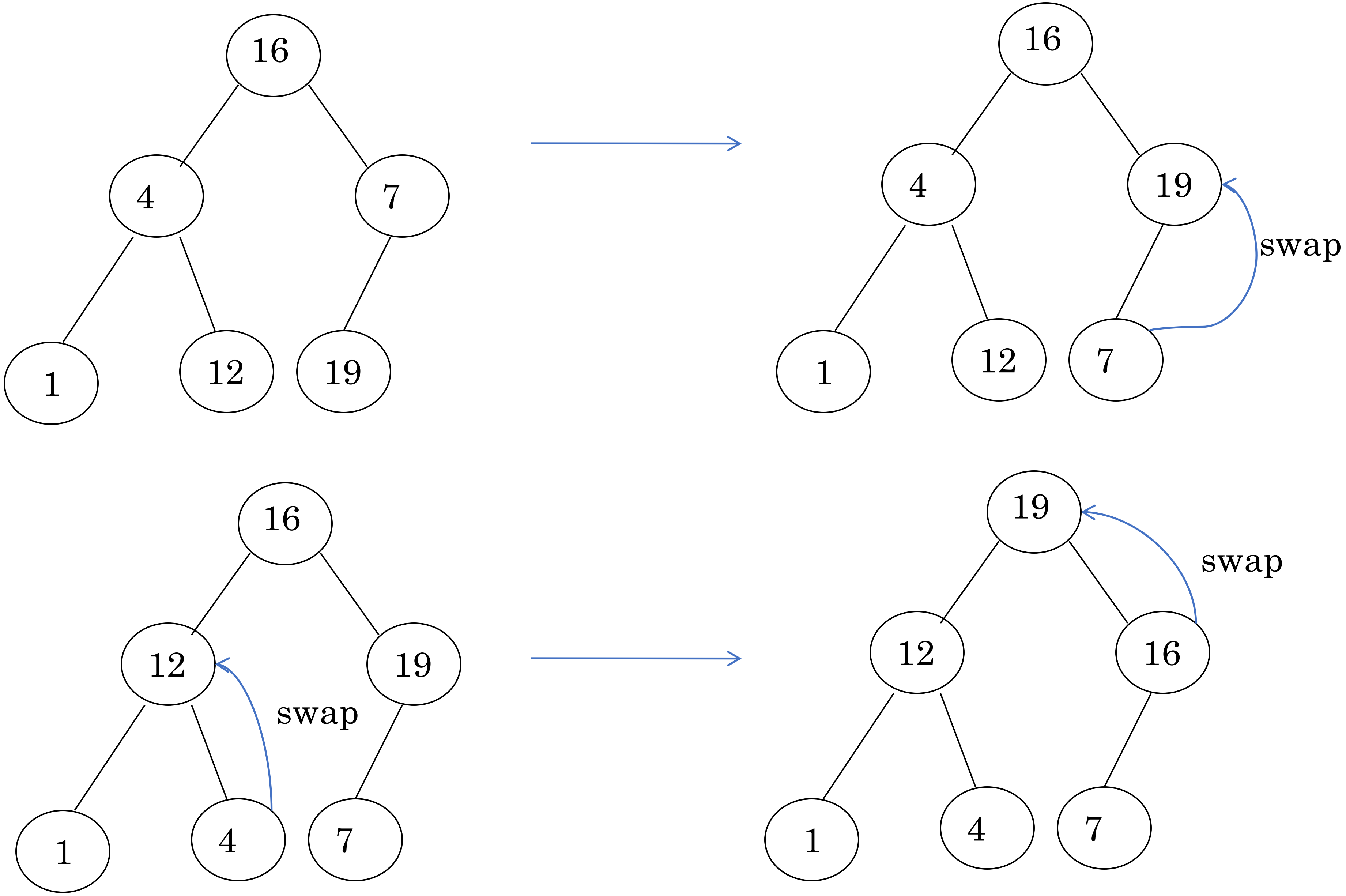
```
Heapsort(A)
{
        Buildheap(A)
        for i ← length[A] //down to 2
            do swap A[1] ←→ A[i]
            A.heapsize ← A.heapsize - 1
            Heapify(A, 1)
}
```

Example: Convert the following array to a heap

16	4	7	1	12	19

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



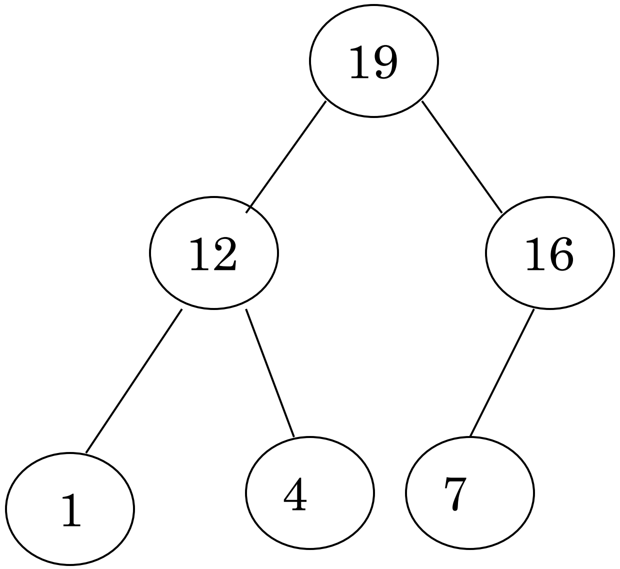


16
4
7
1
12
19
16
4
19
1
12
7
swap
16
12
19
1
4
7
swap
19
12
16
1
4
7
swap

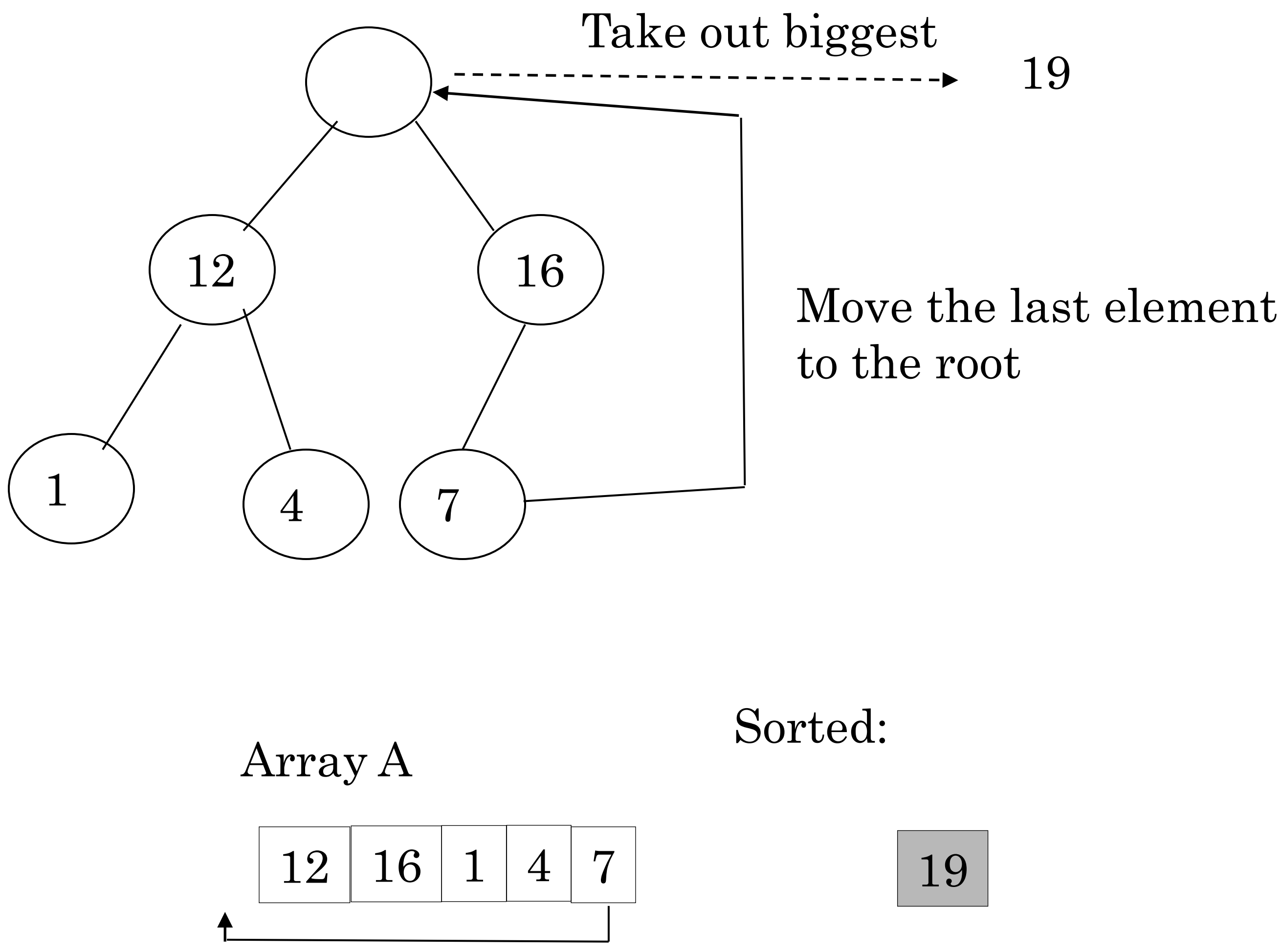
Heap Sort

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

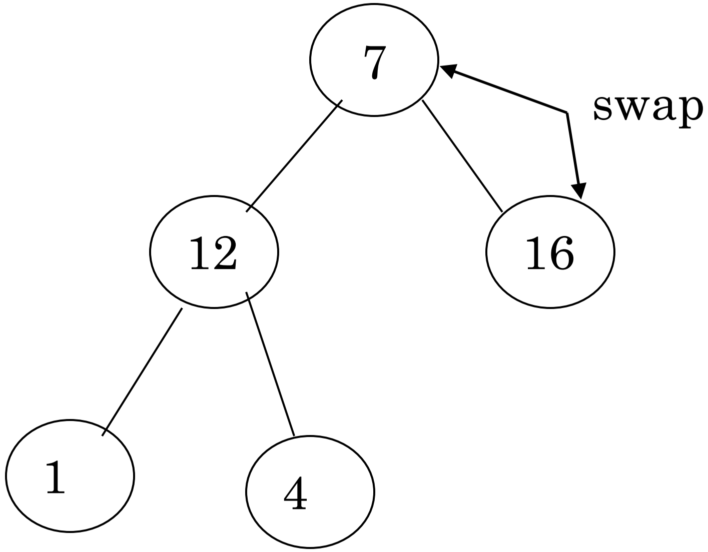
- To sort the elements in the decreasing order, use a min heap
- To sort the elements in the increasing order, use a max heap



Example of Heap Sort



HEAPIFY()

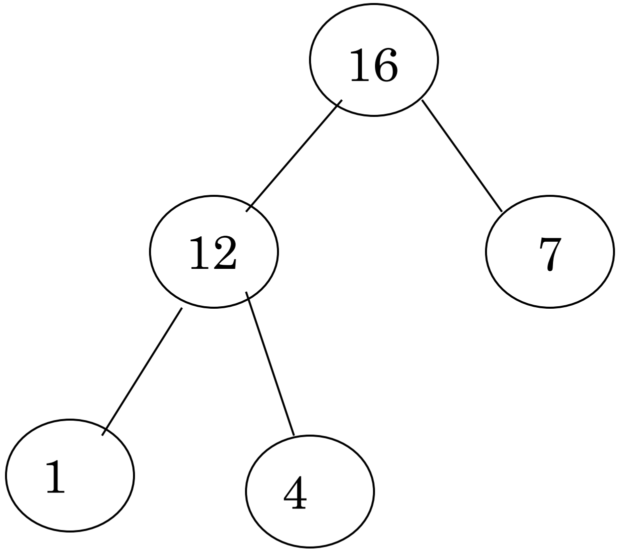


Sorted:

Array A

7	12	16	1	4

19

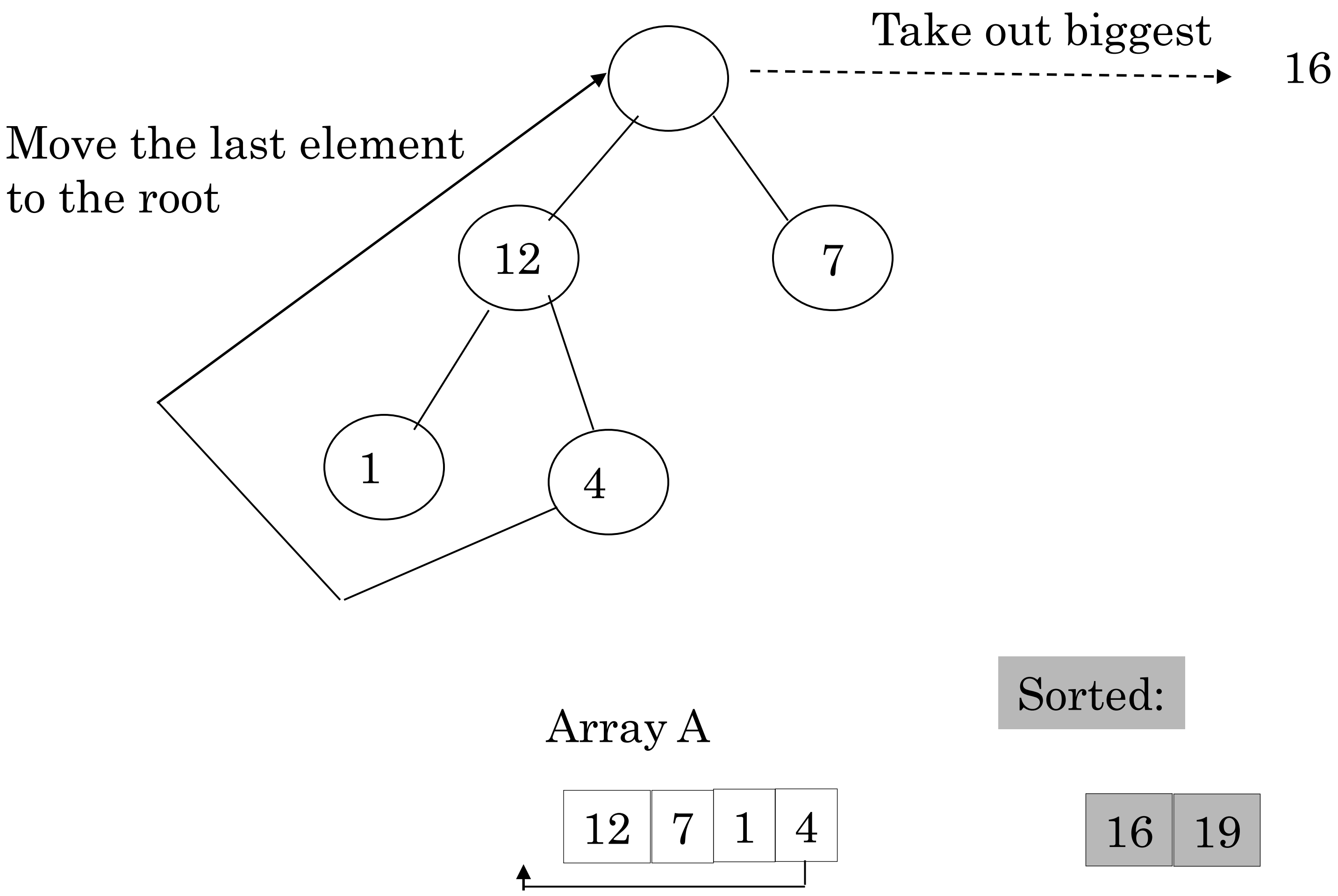


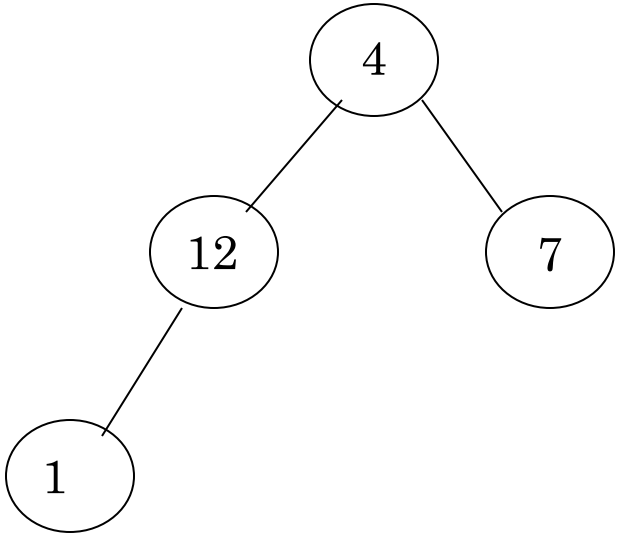
Array A

16	12	7	1	4

Sorted:

19


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



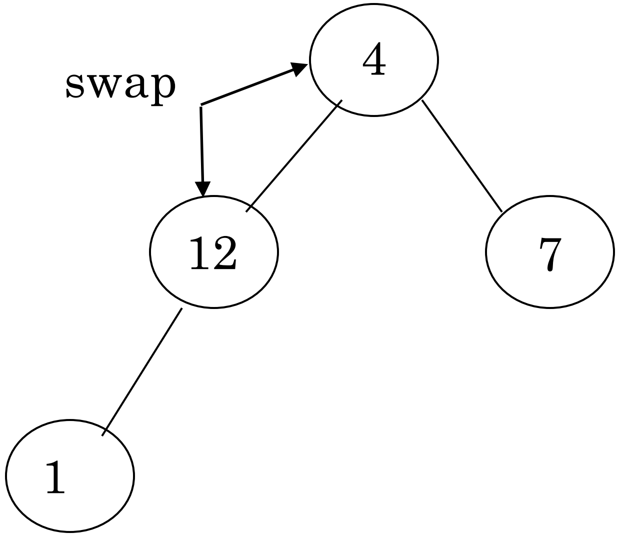
Array A

4	12	7	1

Sorted:

16	19

HEAPIFY()

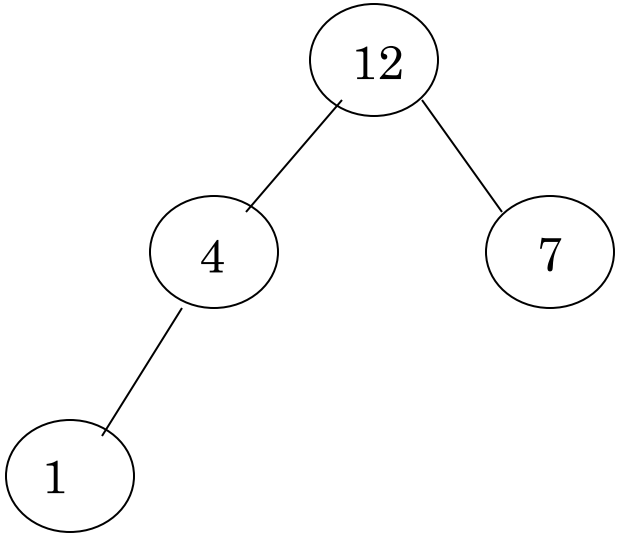


Array A

4	12	7	1

Sorted:

16	19

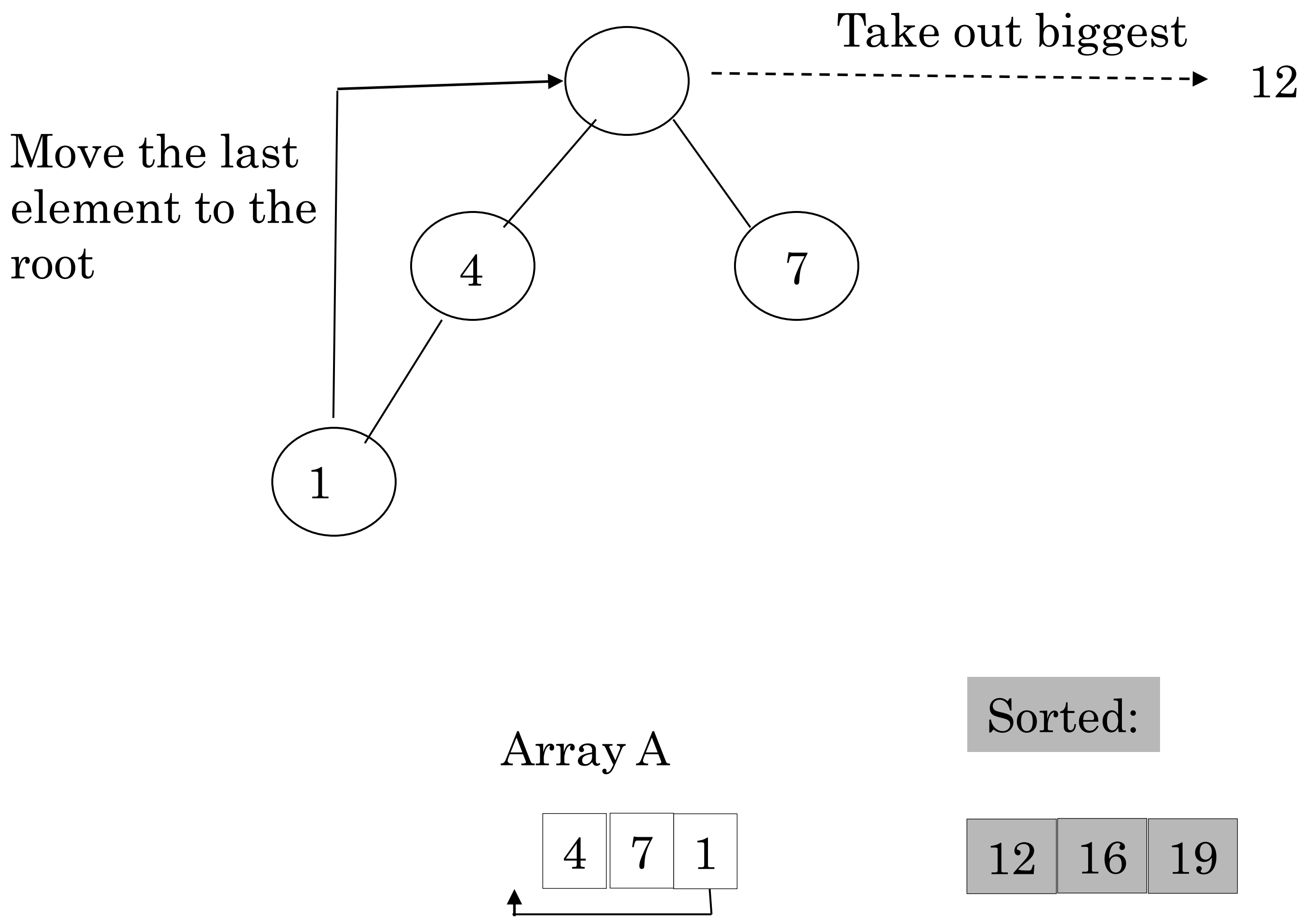


Array A

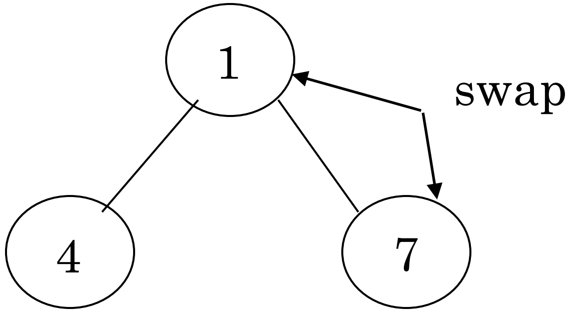
12	4	7	1

Sorted:

16	19



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

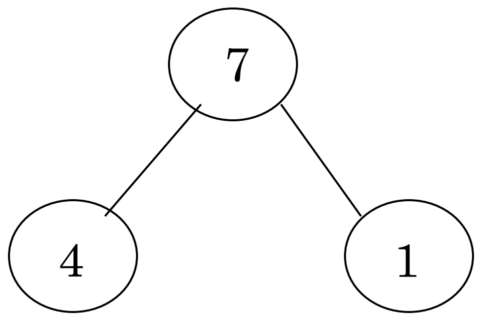


Array A

1	4	7

Sorted:

12	16	19

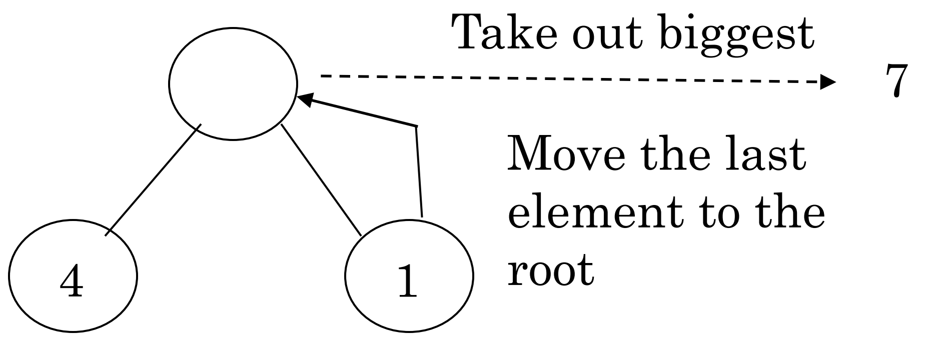


Sorted:

Array A

7	4	1

12	16	19



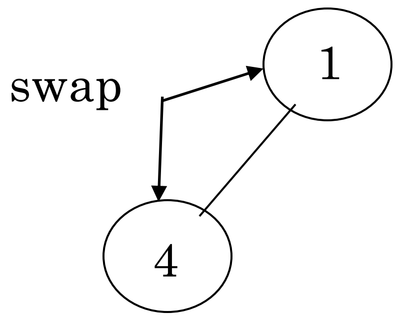
Array A

1	4

Sorted:

7	12	16	19

HEAPIFY()

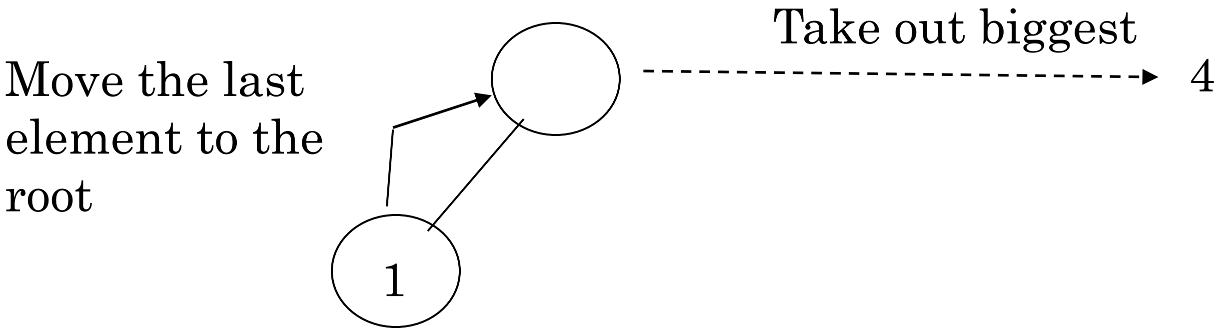


Array A

4	1

Sorted:

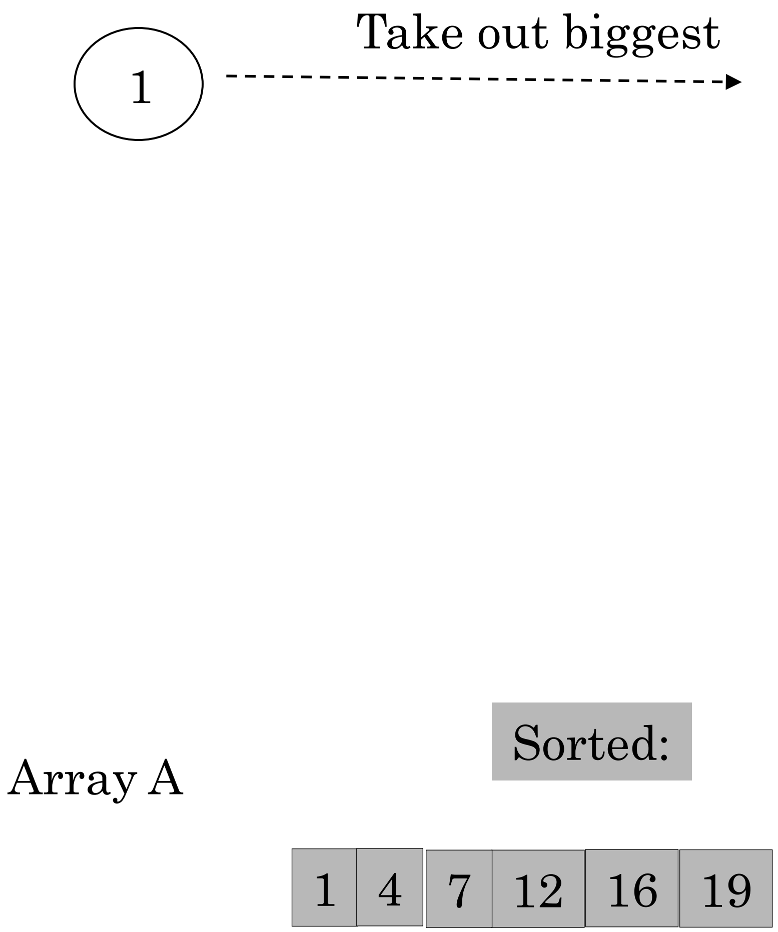
7	12	16	19



Array A

1		4	7	12	16	19

Sorted:


Take out biggest
1
Array A
Sorted:
1 4 7 12 16 19

Sorted:

1	4	7	12	16	19

Complexity of inserting a new node

- Therefore, when we insert a new value in the heap when making the heap, the max number of steps we would need to take comes out to be O(log(n)).
- As we use binary trees, we know that the max height of such a structure is always O(log(n)).
- When we insert a new value in the heap, we will swap it with a value greater than it, to maintain the max-heap property. The number of such swaps would be O(log(n)). Therefore, the insertion of a new value when building a max-heap would be O(log(n)).

Complexity of removing the max valued node from heap

- When we remove the max valued node from the heap, to add to the end of the list, the max number of steps required would also be O(log(n)). Since we swap the max valued node till it comes down to the bottom-most level, the max number of steps we'd need to take is the same as when inserting a new node, which is O(log(n)).

- So total time of Heapify is O(log(n)).

Build-Heap Analysis

- HEAPIFY costs O(lg n) time, and
- Build-HEAP makes O(n) such calls. Thus, the running time is O(n lg n).
- This is a loose upper bound.

Build-Heap Analysis – tight bound

- time for HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Our tighter analysis relies on the properties that an n-element heap has height floor(lgn) and at most ciel(n/2^(h+1)) nodes of any height h.

- The time required by HEAPIFY when called on a node of height h is O(h), and so we can express the total cost of BUILD-HEAP as being bounded from above by

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

- Step 2 uses properties of big-oh notation and ignores the constant. (2^(h+1) = 2.2^h)

Build-Heap Analysis – tight bound

- the upper limit of the summation can be increased to infinity since we are using Big-Oh notation

$$\sum_{h=0}^{\infty} \frac{h}{2^h}$$

- This infinite summation can be solved using:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

for $|x| < 1$.

- We have x = 1/2

Build-Heap Analysis – tight bound

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 .$$

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n) .$$

Heap Sort - Time Analysis

- Total time complexity: $O(n \log n)$

Comparison with Quick Sort and Merge Sort

- Tthe worst-case running time for quick sort is O (n^2), which is unacceptable for large data sets.

- Thus, because of the O(n log n) upper bound on heap sort's running time and constant upper bound on its auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use heap sort.

Comparison with Quick Sort and Merge Sort (cont)

- Heap sort also competes with merge sort, which has the same time bounds, but requires $\Omega(n)$ auxiliary space, whereas heap sort requires only a constant amount. Heap sort also typically runs more quickly in practice.

Possible Application

- When we want to know the task that carry the highest priority given a large number of things to do
- Interval scheduling, when we have a lists of certain task with start and finish times and we want to do as many tasks as possible
- Sorting a list of elements that needs and efficient sorting algorithm

Conclusion

- The primary advantage of the heap sort is its efficiency. The execution time efficiency of the heap sort is O(n log n). The memory efficiency of the heap sort, unlike the other n log n sorts, is constant, O(1), because the heap sort algorithm is not recursive.
- The heap sort algorithm has two major steps. The first major step involves transforming the complete tree into a heap. The second major step is to perform the actual sort by extracting the largest element from the root and transforming the remaining tree into a heap.

Reference

- Deitel, P.J. and Deitel, H.M. (2008) “C++ How to Program”. 6th ed. Upper Saddle River, New Jersey, Pearson Education, Inc.
- Carrano, Frank M. (2007) “Data Abstraction and problem solving with C++: walls and mirrors”. 5th ed. Upper Saddle River, New Jersey, Pearson Education, Inc.