Priority Queues Continued…

Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of add operations
  2. Remove the elements in sorted order with a series of remove_min operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm *PQ-Sort*(*S*, *C*)
  Input sequence *S*, comparator *C* for the elements of *S*
  Output sequence *S* sorted in increasing order according to *C*
  P ← priority queue with comparator *C*
  while ¬*S.is_empty* ()
    e ← *S.remove_first* ()
    P.add (*e*, ∅)
  while ¬*P.is_empty*()
    e ← *P.removeMin*().*key*()
    S.add_last(*e*)

Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of add operations
  2. Remove the elements in sorted order with a series of remove_min operations
- The running time of this sorting method depends on the priority queue implementation, if maxHeap is used then it is NlogN

Algorithm *PQ-Sort*(*S*, *C*)
Input sequence *S*, comparator *C* for the elements of *S*
Output sequence *S* sorted in increasing order according to *C*
P ← priority queue with comparator *C*
while ¬*S.is_empty* ()
e ← *S.remove_first* ()
P.add (*e*, ∅)
while ¬*P.is_empty*()
e ← *P.removeMin*().*key*()
S.add_last(*e*)

PQSort

```
def pq sort(C):
"""Sort a collection of elements stored in a positional list."""
    n = len(C)
    P = PriorityQueue()
    for j in range(n):
        element = C.delete(C.first())
        P.add(element, element) # use element as key and value
    for j in range(n):
        (k,v) = P.remove min()
        C.add last(v)  # store smallest remaining element in C
```

An implementation of pq_sort function assuming an appropriate implementation of a PriorityQueue class. Note that each element of the input list C serves as its own key in the priority queue P.

Heap Sort

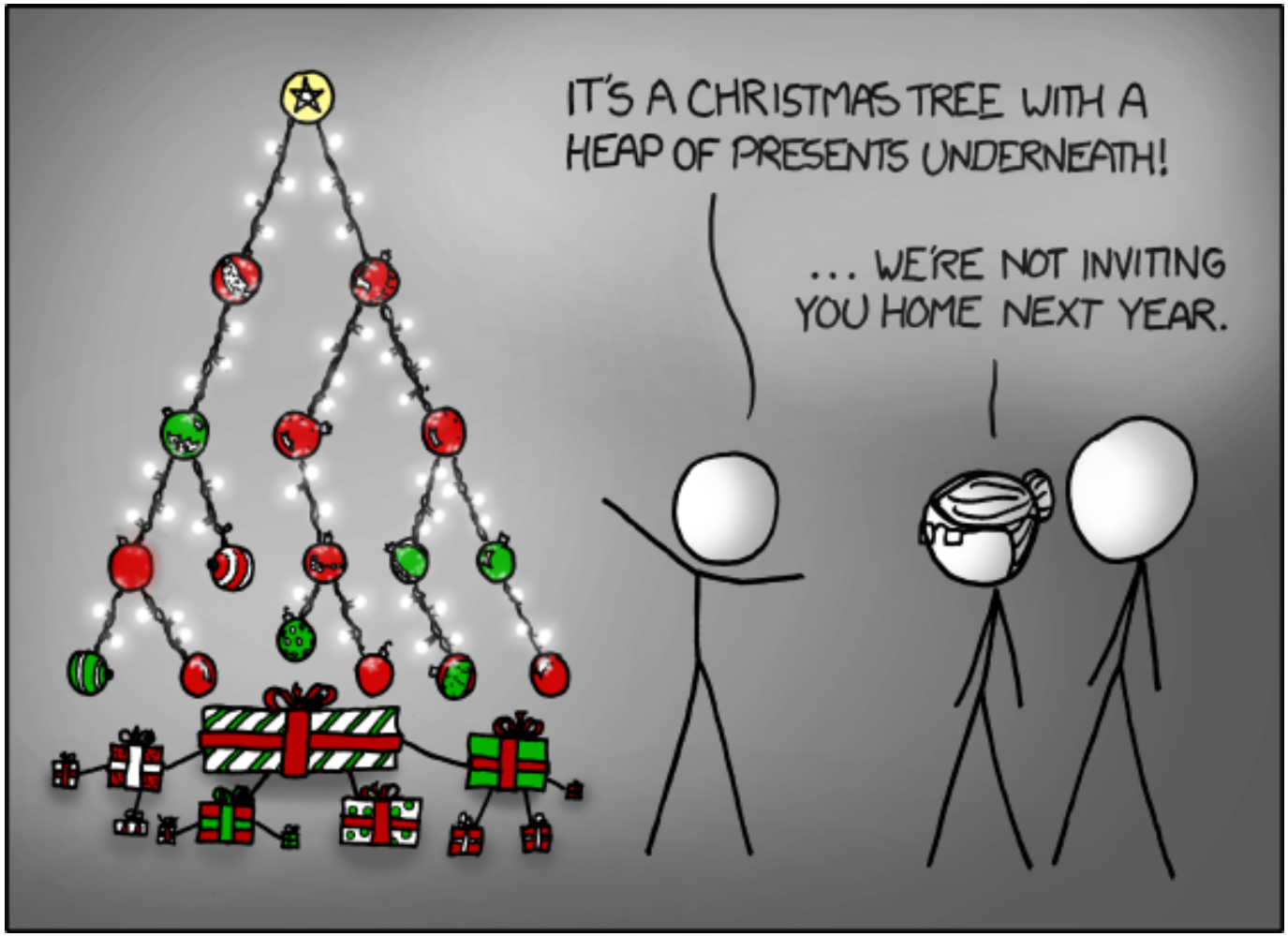
How to implement Heap Sort

1. Heapsort begins by heapifying the array into a **max-heap** and initializing an end index value to the size of the array minus 1.
2. Heapsort
   I. **repeatedly removes the maximum value**,
   II. stores that value at the end index
   III. decrements the end index.
   IV. The removal loop repeats until the end index is 0.

What is a max heap?

Nerdy Example!

Heaps!

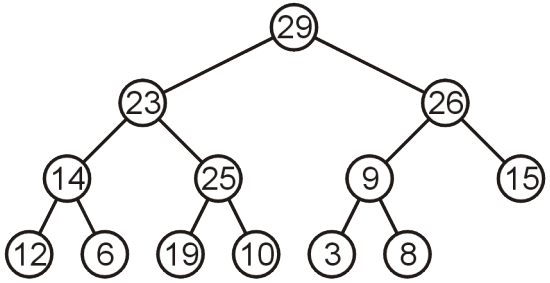


http://xkcd.com/835/

Binary Max Heaps

A binary max-heap is identical to a binary min-heap except that the **parent is always larger than either of the children**

For example, the same data as before stored as a max-heap in this array yields to following tree. Please note that 0^{th} index left empty to simplify the the formula for finding parent, left child and right child.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	29	23	26	14	25	9	15	12	6	19	10	3	8		

How to convert an array filled with values into a max heap?

Example 1

Converting the array into a max heap

Let us look at this example: **we must convert the unordered array** with $n = 10$ elements into a max-heap

46	52	28	17	3	63	34	81	70	95

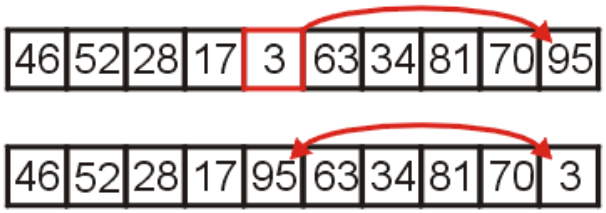
None of the leaf nodes need to be percolated down, and the first non-leaf node is in position $n/2$

And the reason we select the middle element is if you draw this array as a binary tree, you will notice that middle element is the last parent of this tree with its potential left and right children.

Thus we start with position 10/2 = 5, position 5 is at index 4

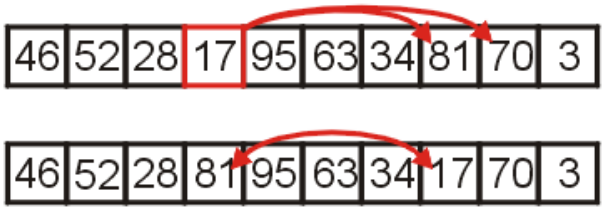
Converting the array into a max heap

We compare the value **3** with its child and swap them.3 and 95 is now swamped



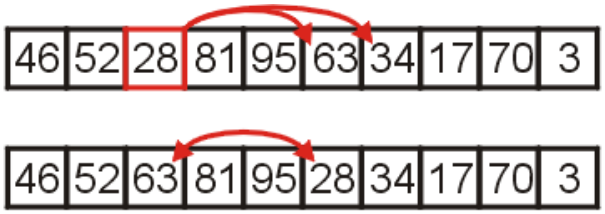
Converting the array into a max heap

We compare **17** with its two children and swap it with the maximum child (70)



Converting the array into a max heap

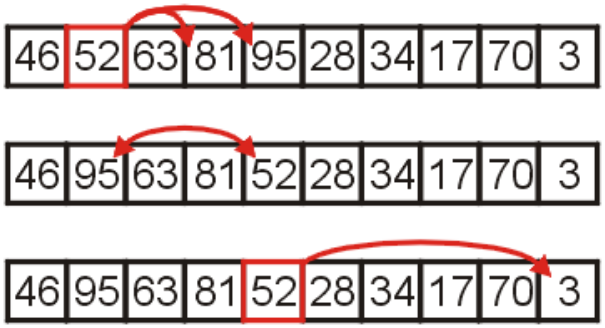
We compare **28** with its two children, 63 and 34, and swap it with the largest child



Converting the array into a max heap

We compare **52** with its children, swap it with the largest

- Recursing, no further swaps are needed

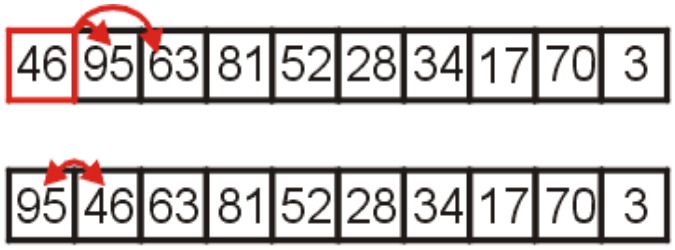


Converting the array into a max heap

Finally, we swap the root with its largest child, and recurse, swapping 46 again with 81, and then again with 70

Converting the array into a max heap

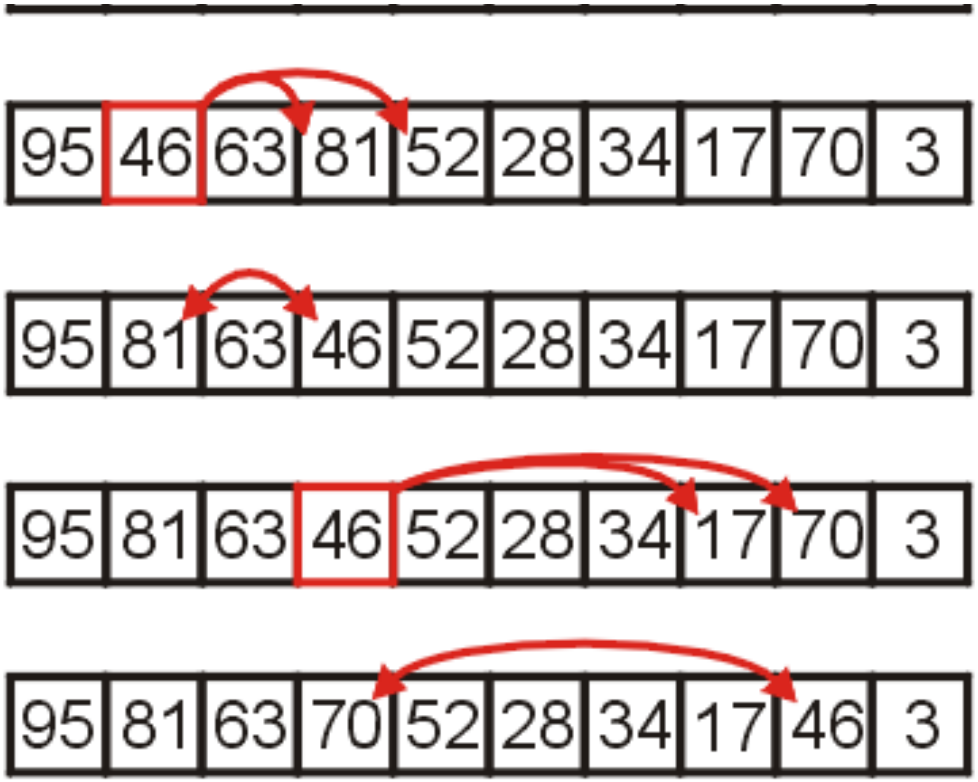
Finally, we swap the root with its largest child, and recurse, swapping 46 again with 81, and then again with 70



Once we have 95 in its place, we now must ensure 46 is in the right location.

Converting the array into a max heap

Finally, we swap the root with its largest child, and recurse, swapping 46 again with 81, and then again with 70



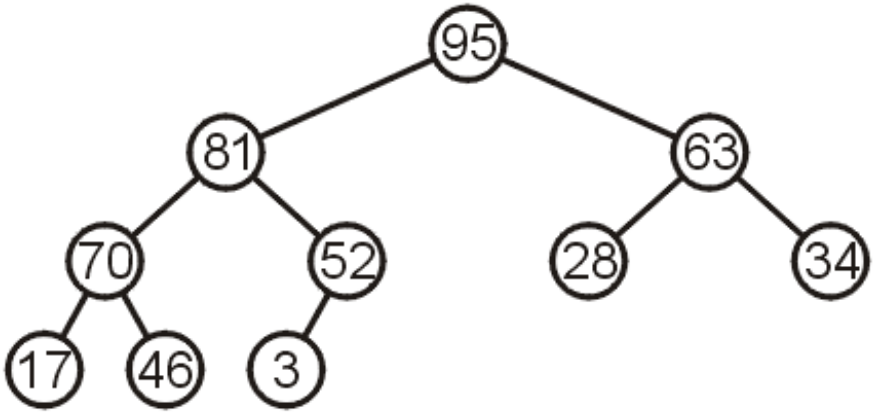
Converting the array into a max heap

We have now converted the unsorted array

46	52	28	17	3	63	34	81	70	95

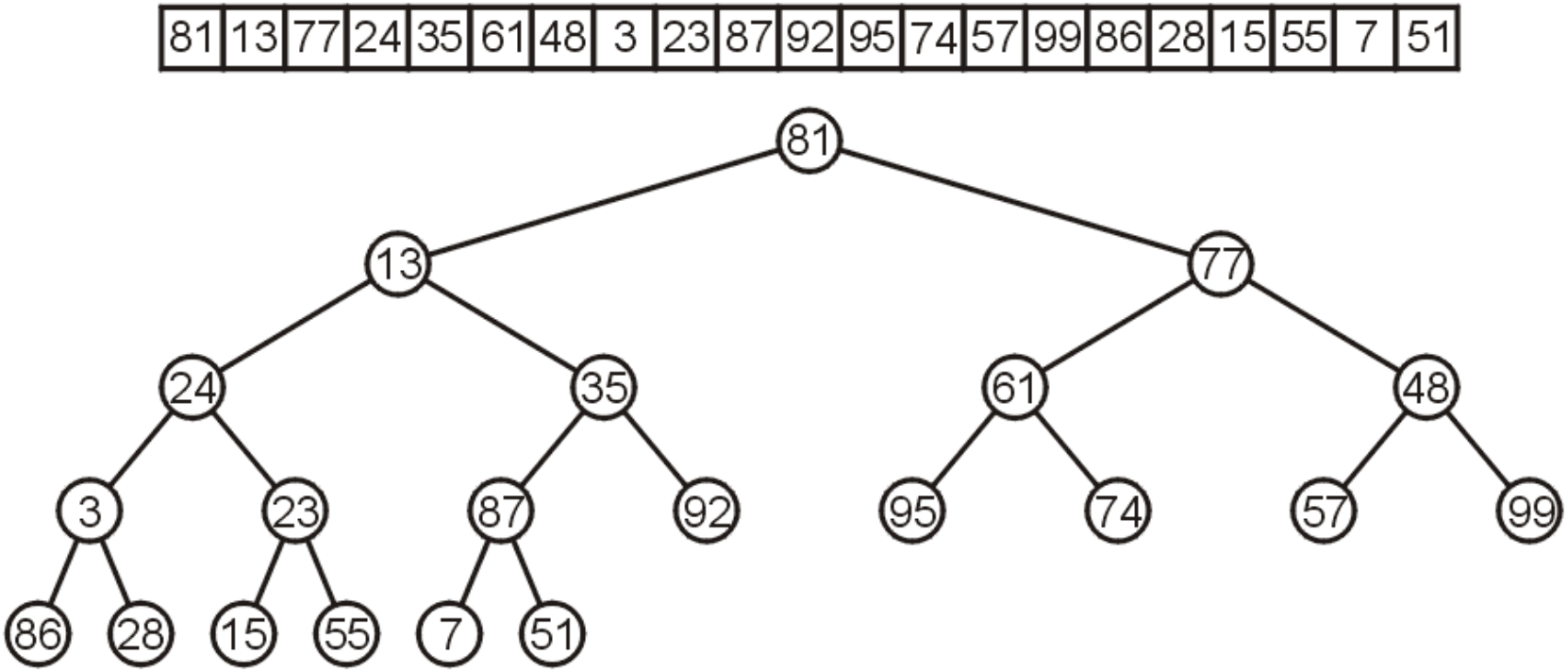
into a max-heap: below is the resulting array and the tree (note that middle element is residing at index 4 which is 52 and as you see that is the last parent in the tree with only left child)

95	81	63	70	52	28	34	17	46	3



Example 2

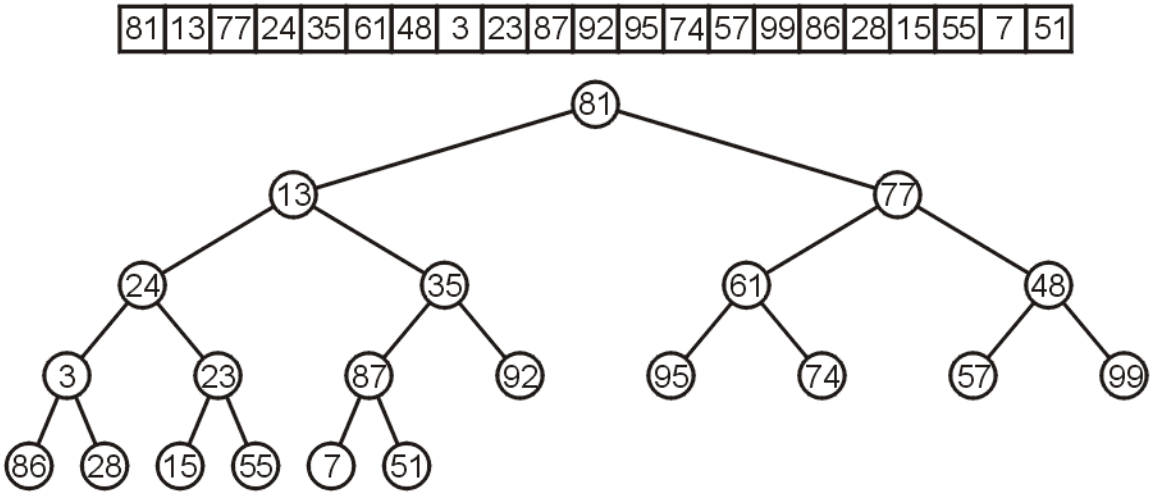
Converting the array into a max heap



Converting the array into a max heap

Each leaf node is a max heap on its own

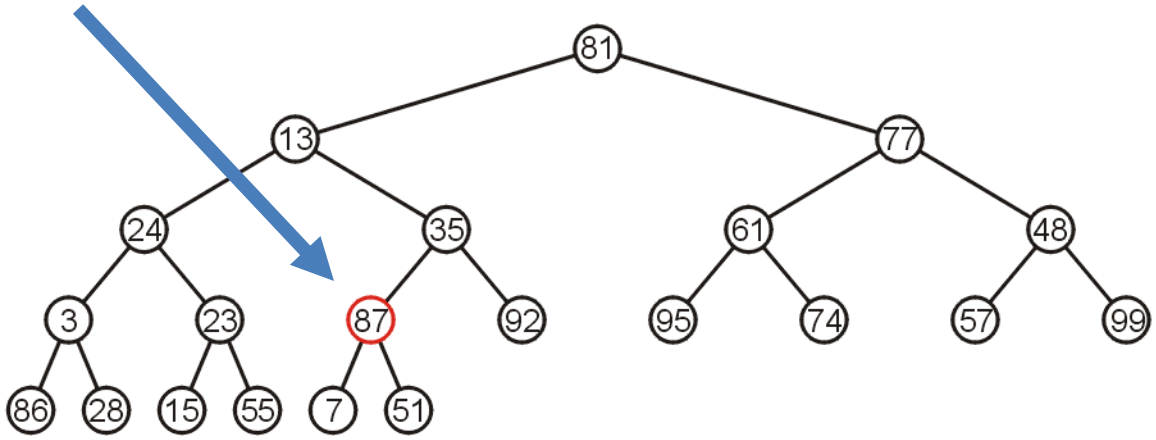
See given array below, goal is to convert it to max heap. Find the mid point to start which points to the location with 87. That is our starting point!



Converting the array into a max heap

We note that all leaf nodes are trivial heaps

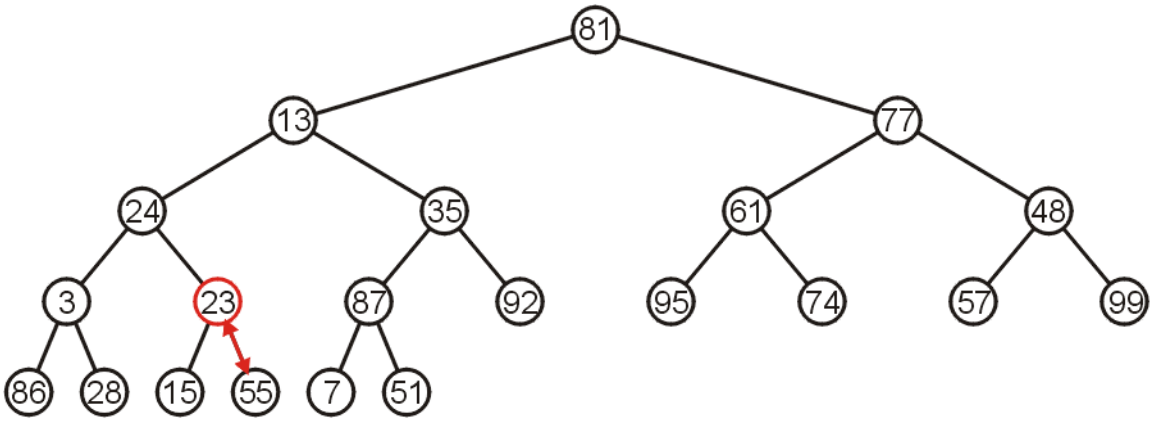
Also, the sub-tree with 87 as the root is a max-heap



Converting the array into a max heap

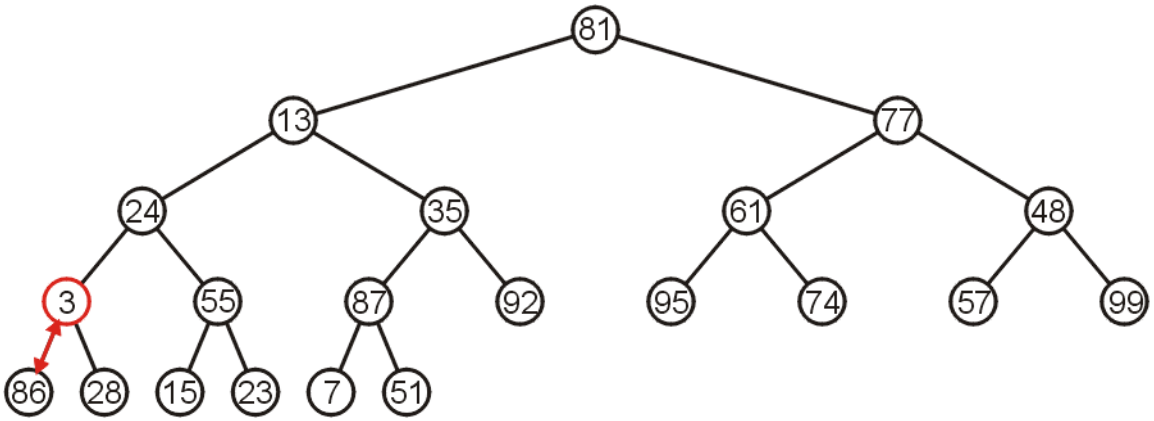
The sub-tree with 23 is not a max-heap, but swapping it with 55 creates a max-heap

This process is termed ***percolating down***



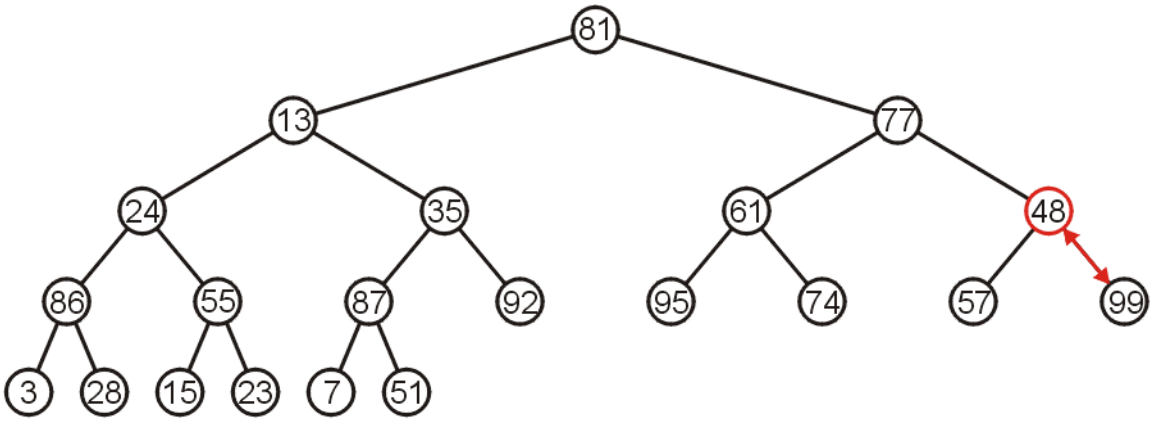
Converting the array into a max heap

The sub-tree with 3 as the root is not max-heap, but we can swap 3 and the maximum of its children: 86



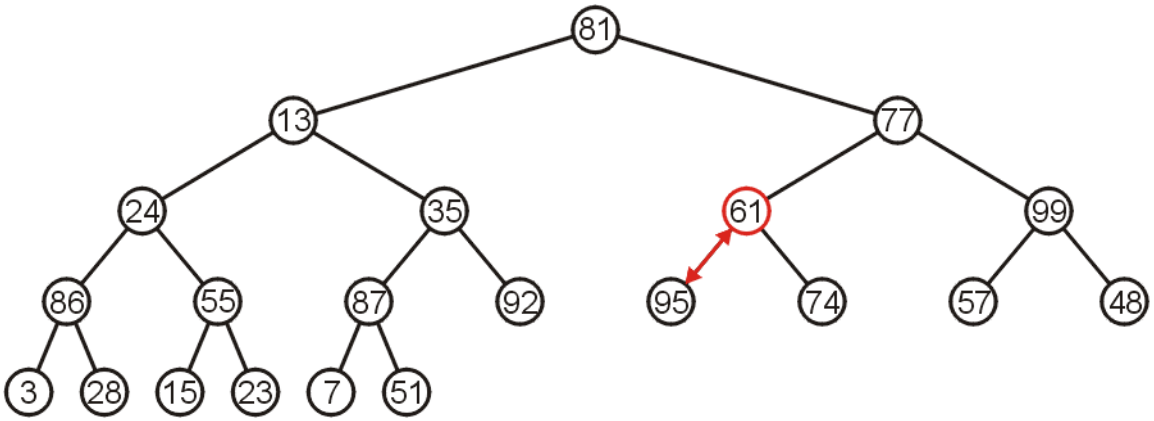
All this process is also knowm as In-place Heapification

Starting with the next higher level, the sub-tree with root 48 can be turned into a max-heap by swapping 48 and 99



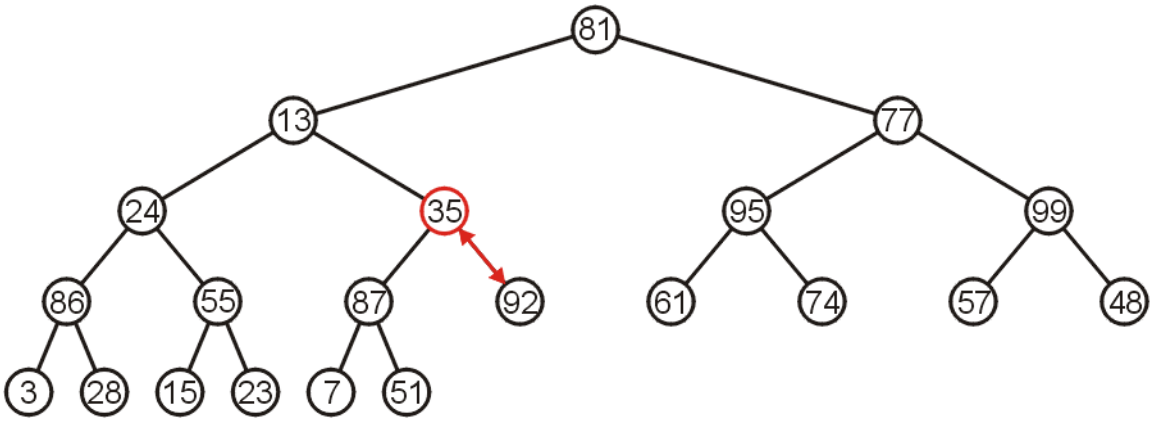
In-place Heapification

Similarly, swapping 61 and 95 creates a max-heap of the next sub-tree



In-place Heapification

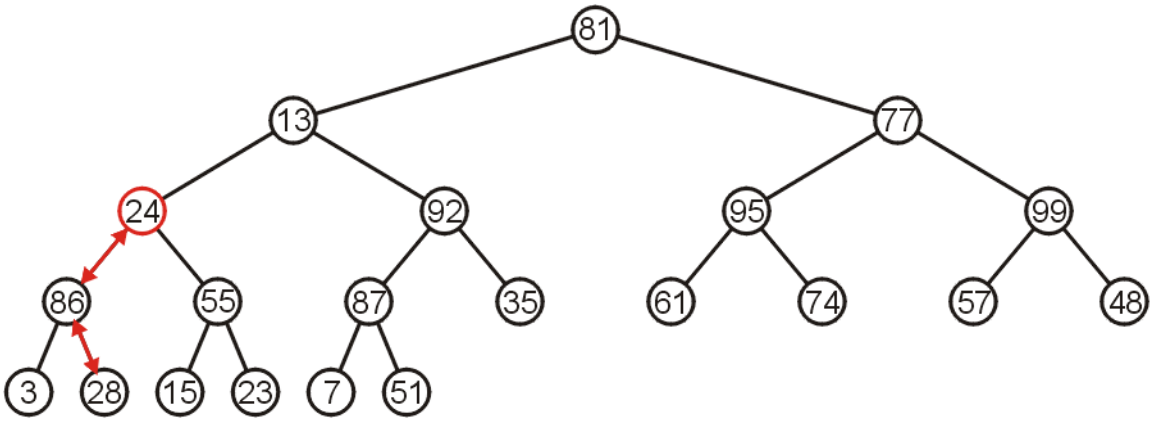
As does swapping 35 and 92



In-place Heapification

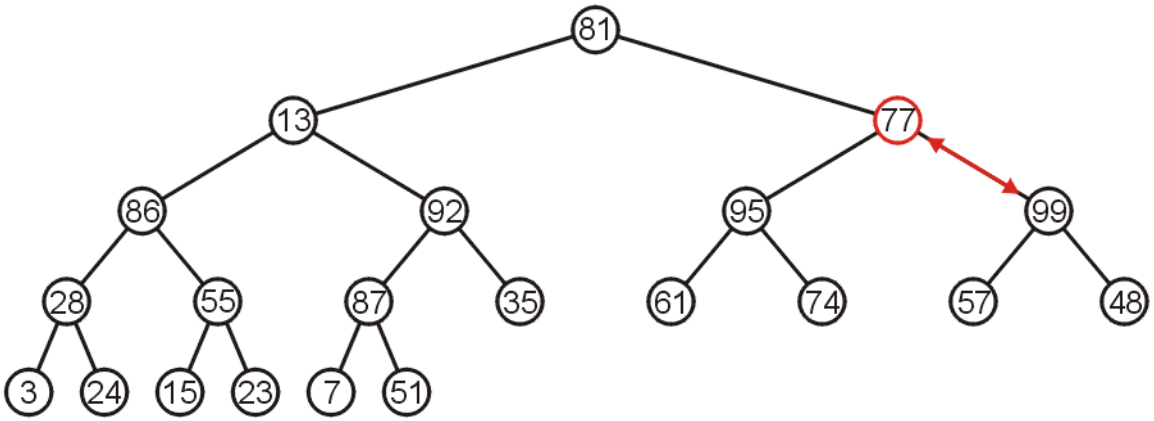
The sub-tree with root 24 may be converted into a max-heap by

1. **first swapping 24 and 86 and**
2. **then swapping 24 and 28**



In-place Heapification

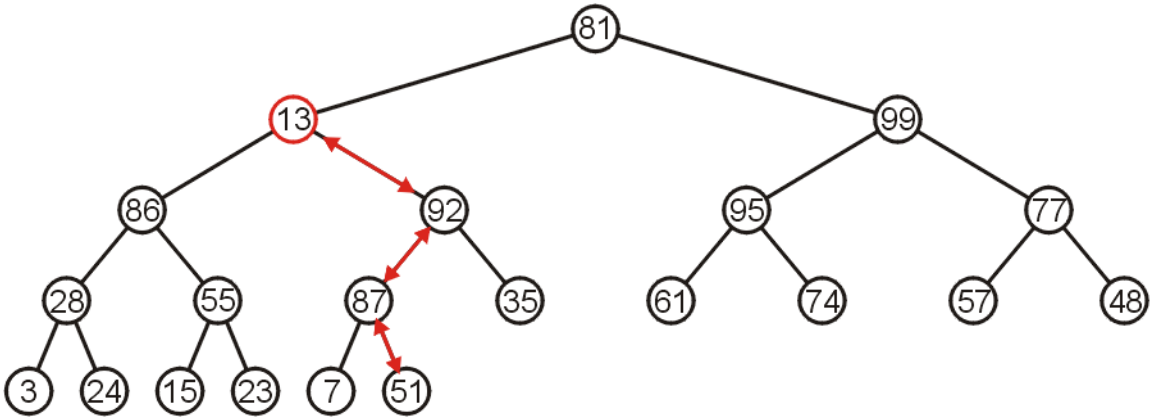
The right-most sub-tree of the next higher level may be turned into a max-heap by swapping 77 and 99



In-place Heapification

However, to turn the next sub-tree into a max-heap requires that 13 be percolated down to a leaf node.

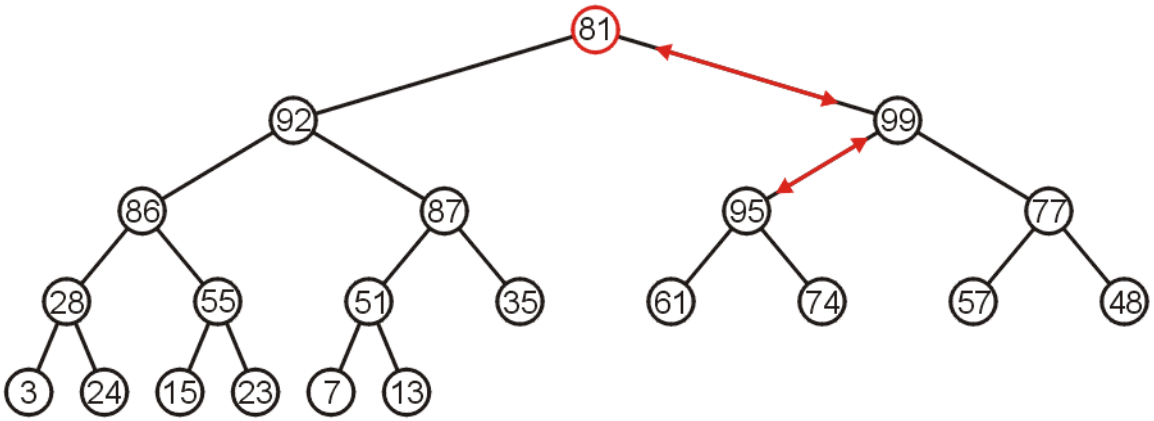
1. **First swapping 13 with 92**
2. **Then swapping 13 with 87**
3. **Then swapping 13 with 51**



In-place Heapification

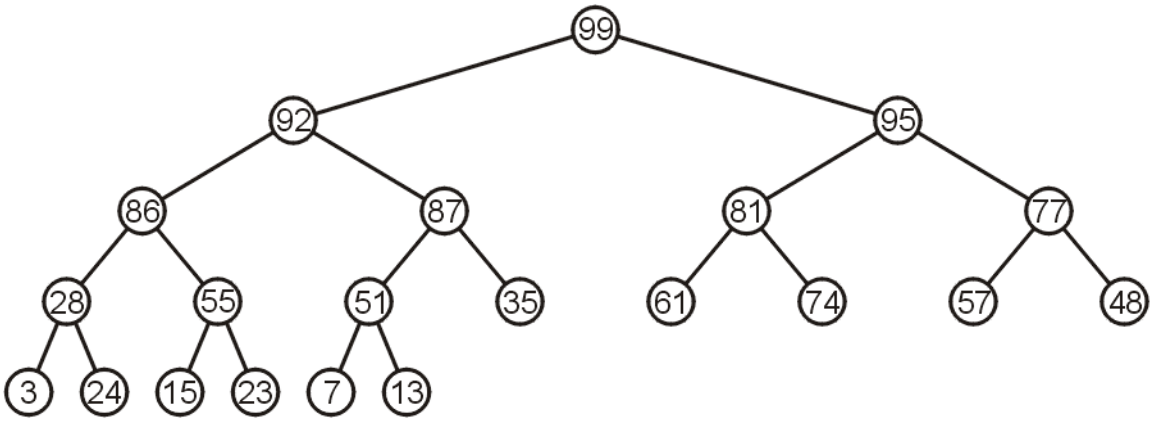
The root need only be percolated down by two levels

1. **First swapping 81 with 99**
2. **Then swapping 81 with 95**



In-place Heapification

The final product is a max-heap



Heap sort

a.k.a. *heapsort*

How to implement HeapSort

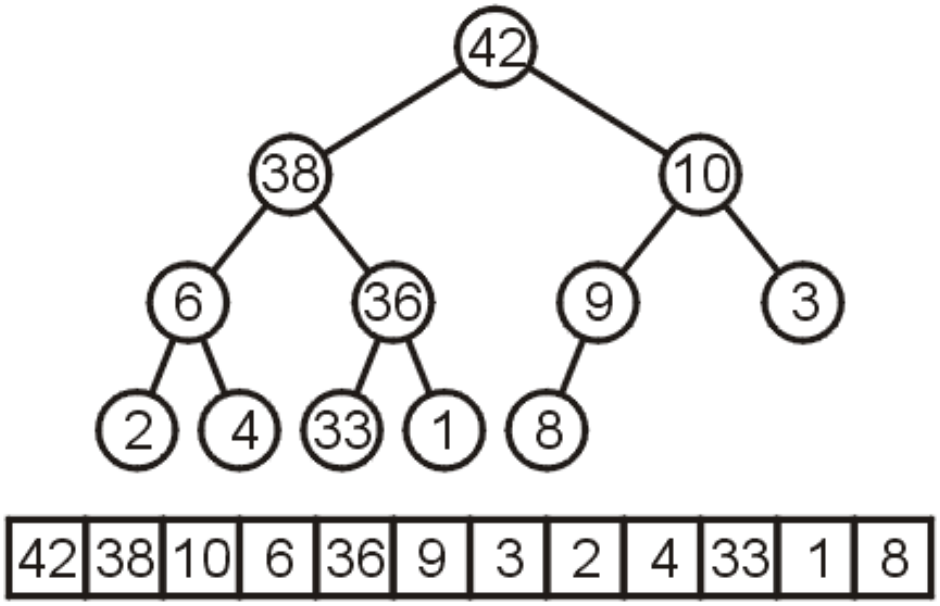
1. Heapsort begins by heapifying the array into a **max-heap** and initializing an end index value to the size of the array minus 1.
2. Heapsort
   I. **repeatedly removes the maximum value**,
   II. stores that value at the end index
   III. decrements the end index.
   IV. The removal loop repeats until the end index is 0.

Implementing Heap Sort in place

In-place Implementation

Instead of implementing a min-heap, consider a max-heap:

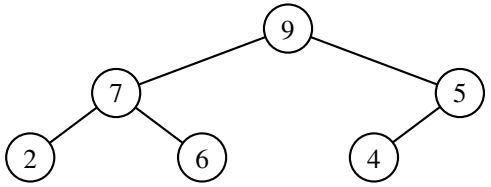
- A heap where the maximum element is at the top of the heap and the next to be popped



In place Heap Sort, very simple example

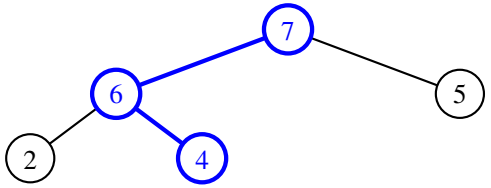
(a)

9	7	5	2	6	4



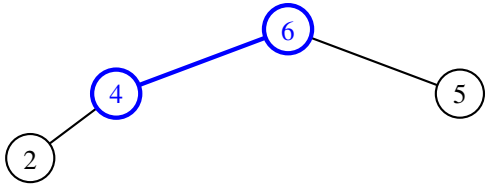
(b)

7	6	5	2	4	9



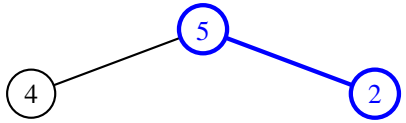
(c)

6	4	5	2	7	9



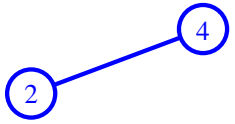
(d)

5	4	2	6	7	9



(e)

4	2	5	6	7	9



(f)

2	4	5	6	7	9

Outline

Heap Sort is an $O(n \log(n))$ sorting algorithm

We will:

- define the strategy
- analyze the run time
- convert an unsorted list into a heap
- cover some examples

Bonus: performed in place

In-place Implementation

Is it possible to perform a heap **sort in place**, that is, require at most $\Theta(1)$ memory (a few extra variables)?

Implementation of HeapSort

Heap Sort

- Heaps can be used in sorting an array.
- In max-heaps, maximum element will always be at the root. Heap Sort uses this property of heap to sort the array.
- Consider an array which is to be sorted using Heap Sort.
- Initially build a max heap of elements in this array.
- The root element will contain maximum element of
- After that, swap this element with the last element of this array and heapify the max heap excluding the last element which is already in its correct position and then decrease the length of heap by one.
- Repeat the step 2, until all the elements are in their correct position.

Heap Sort Algorithm

```
Heapsort(numbers, numbersSize) {
   // Heapify numbers array
   for (i = numbersSize / 2 - 1; i >= 0; i--)
      MaxHeapPercolateDown(i, numbers, numbersSize)

   for (i = numbersSize - 1; i > 0; i--) {
      swap numbers[0] and numbers[i]
      MaxHeapPercolateDown(0, numbers, i)
   }
}
```

This Heapsort algorithm uses **2 loops to sort an array**.

1. The first loop heapifies the array using **MaxHeapPercolateDown**.
2. The second loop removes the maximum value, stores that value at the end index, and decrements the end index, until the end index is 0.

Example of Heap Sort

Example Heap Sort

Let us look at this example: we must first convert the unordered array with $n = 10$ elements into a max-heap (please note that we completed this exercise as Example 1 in earlier slides, so we do have the result as max heap in following slide)

46	52	28	17	3	63	34	81	70	95

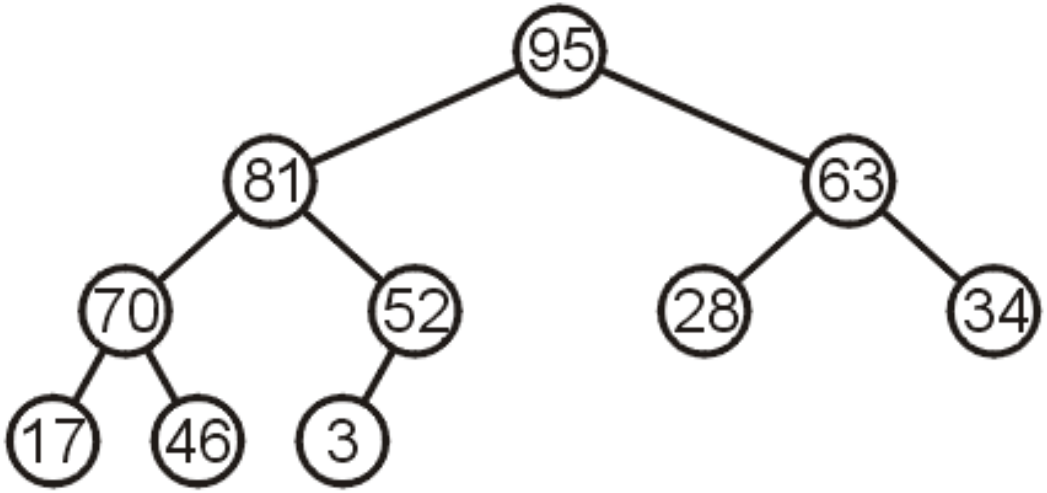
Heap Sort Example

We have now converted the **unsorted array**

46	52	28	17	3	63	34	81	70	95

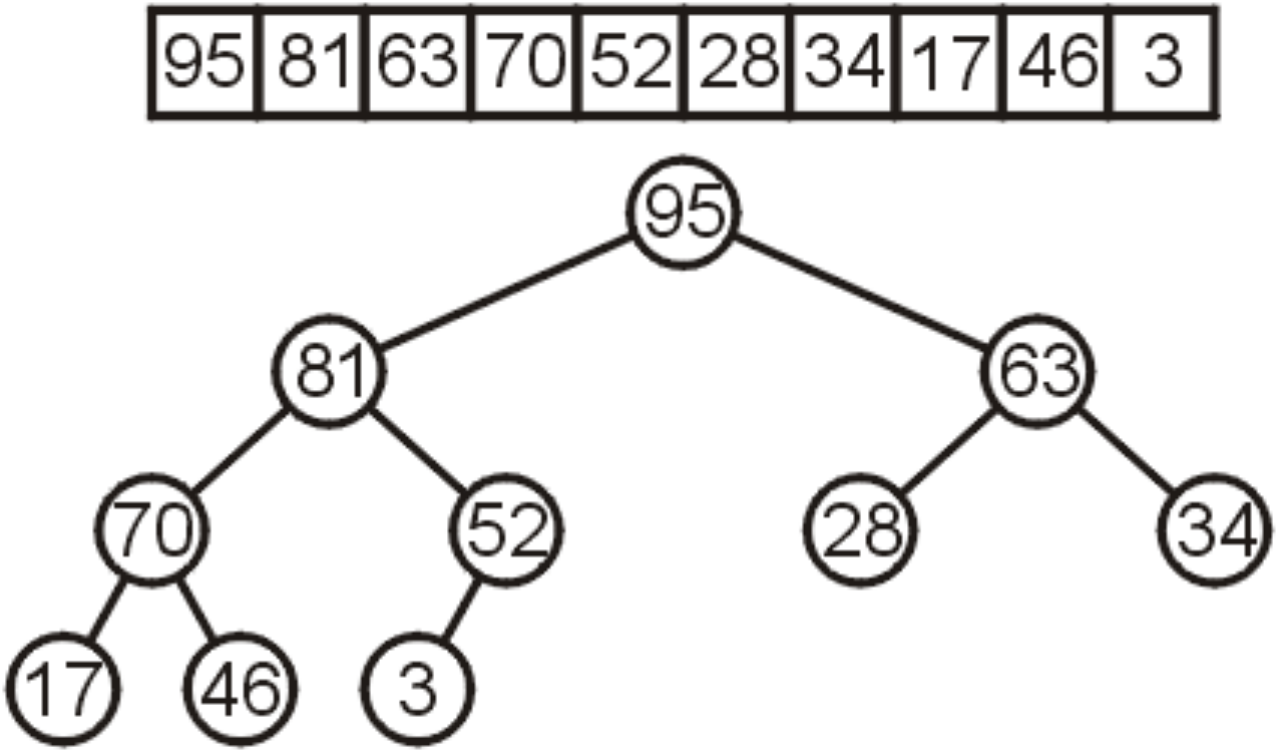
into a max-heap:

95	81	63	70	52	28	34	17	46	3



Now that we have a max heap
We need to sort in place using
Heapsort Algorithm described earlier

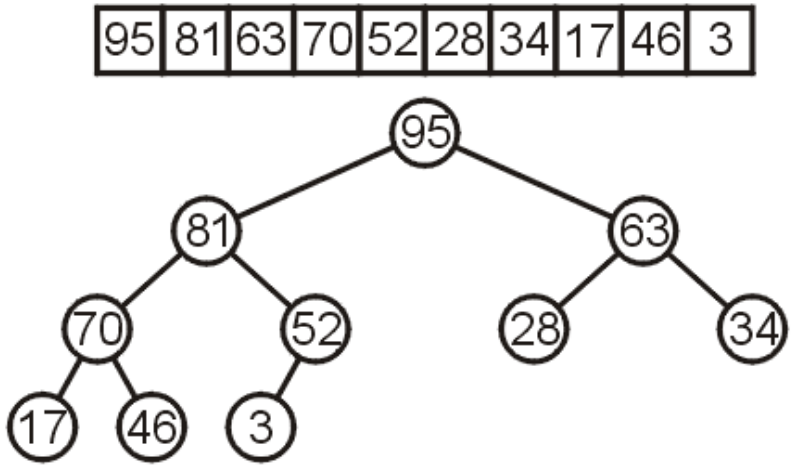
Max Heap



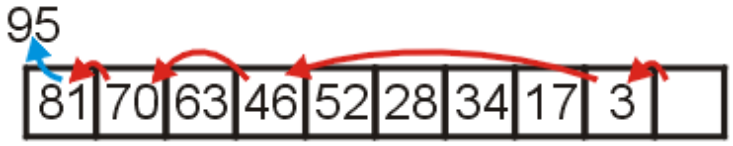
Max heaps creates data with the largest value at the root! That will help us move the elements

Using Heap Sort on Max Heap

Consider popping the maximum element of this heap…which is 95 and swap it with 3, then percolate 3 down to ensure it still is a max heap

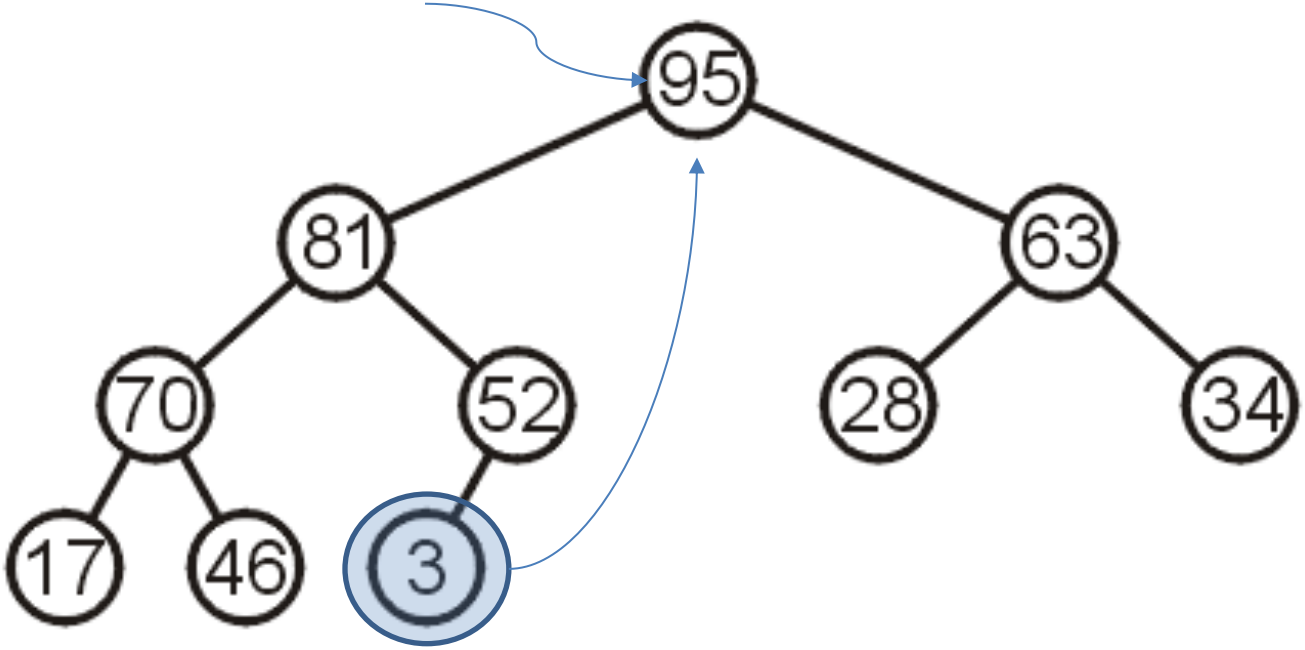


After removing 95 and replacing it with 3, we percolated 3 down and that resulted in following max heap, where should 95 go?



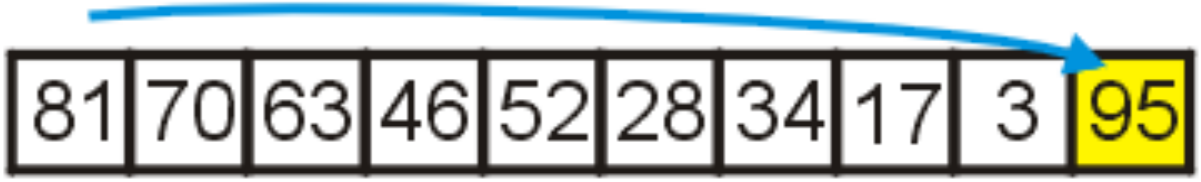
Where did 95 go ?

- Here is what happened? We popped 95 , we swapped 95 and 3… 3 was placed where 95 used to be, then we must apply percolate down to make 3 is in its right place in order remain as a max heap.



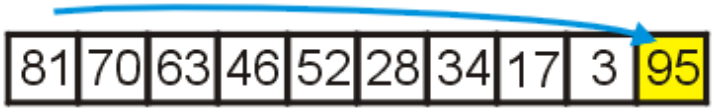
Where did 95 go ?

We percolated 3 down and following shows our max heap with 95 located at the last index.

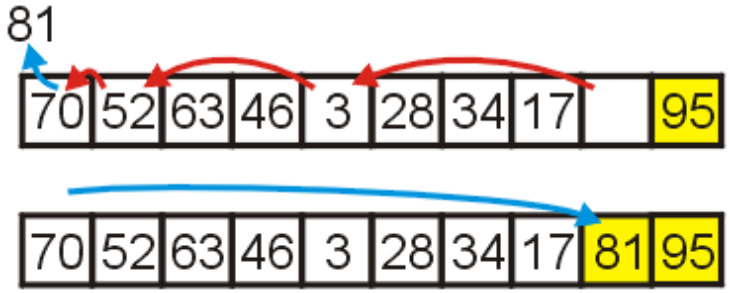


Heap Sort Example

This is the last entry in the array, so why not fill it with the largest element?



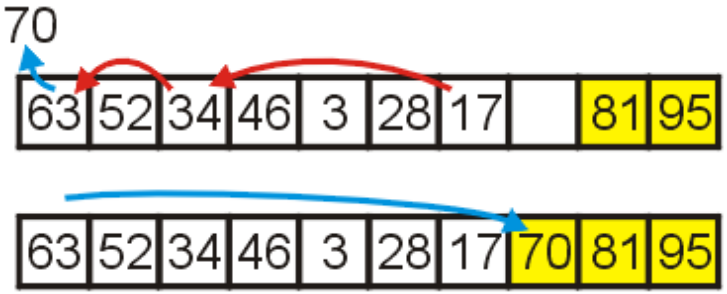
Repeat this process: pop the maximum element, place 3 in where 81 used to be and percolate down …



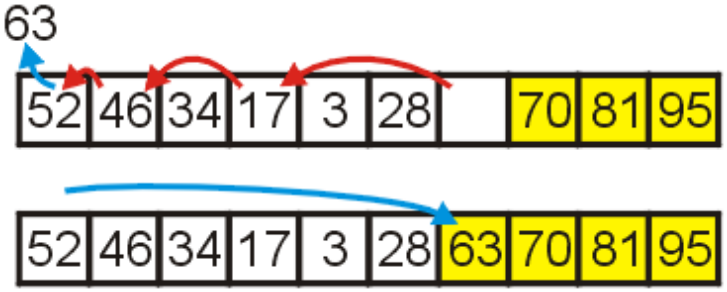
Heap Sort Example

Repeat this process

– Pop and append 70



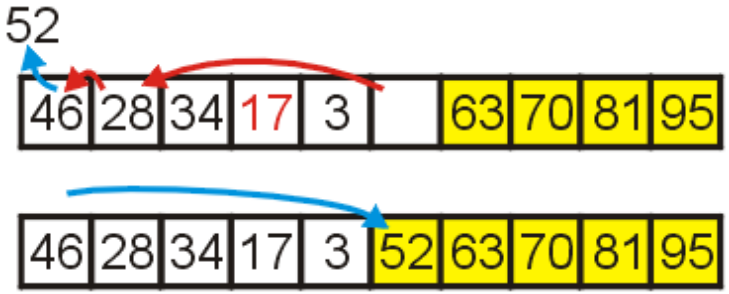
– Pop and append 63



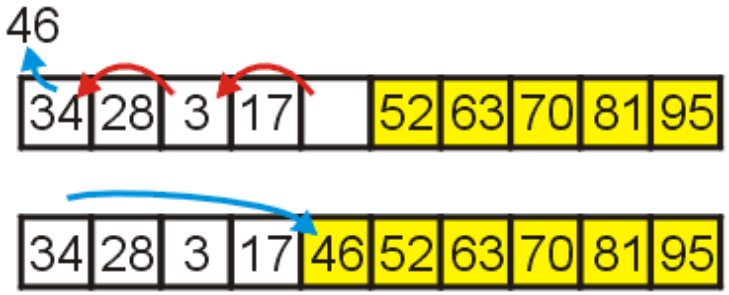
Heap Sort Example

We have the 4 largest elements in order

- Pop and append 52



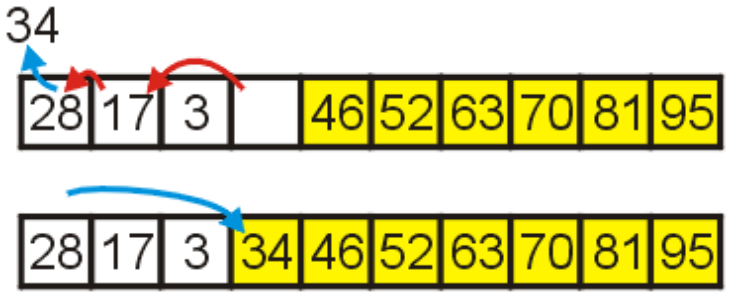
- Pop and append 46



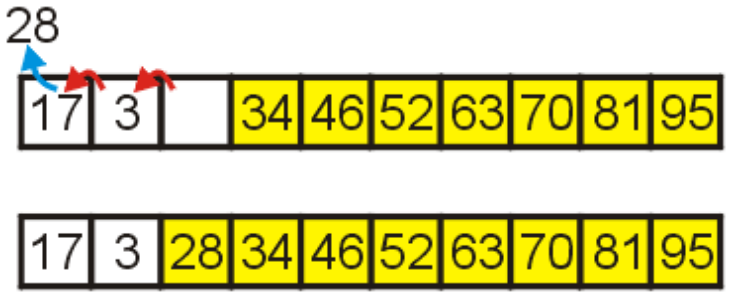
Heap Sort Example

Continuing...

– Pop and append 34

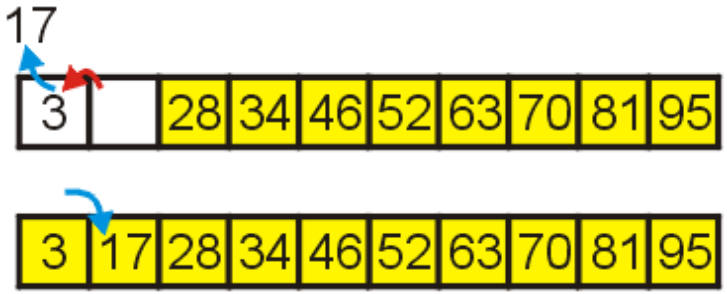


– Pop and append 28



Heap Sort Example

Finally, we can pop 17, insert it into the 2nd location, and the resulting array is sorted



Heap Sort Analysis
O(nlogn) sorting algorithm

***Reference**: Introduction to Algorithms 3rd edition, by Cormen, Leiserson, Rivest and Stein*
Chapter 6 , Appendix A.6 and A.8

Heap Sort Pseudocode (CLRS)

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Let’s discuss each function written in heapsort pseudocode

MAX-HEAPIFY(A,i)

Pseudocode for MAX-HEAPIFY.

- In order to maintain the max-heap property, we call the procedure MAX-HEAPIFY.
- Its inputs are an array A and an index i into the array.
- When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that A[i] might be smaller than its children, thus violating the max-heap property.
- MAX-HEAPIFY lets the value at A[i] **"float down" or also known as "percolate down"** in the max-heap so that the sub-tree rooted at index i obeys the max-heap property.

Please note that in CLRS array indexing starts from 1.

So Array A is defined as A[1..A.length]

Example of a Max Heap Violation

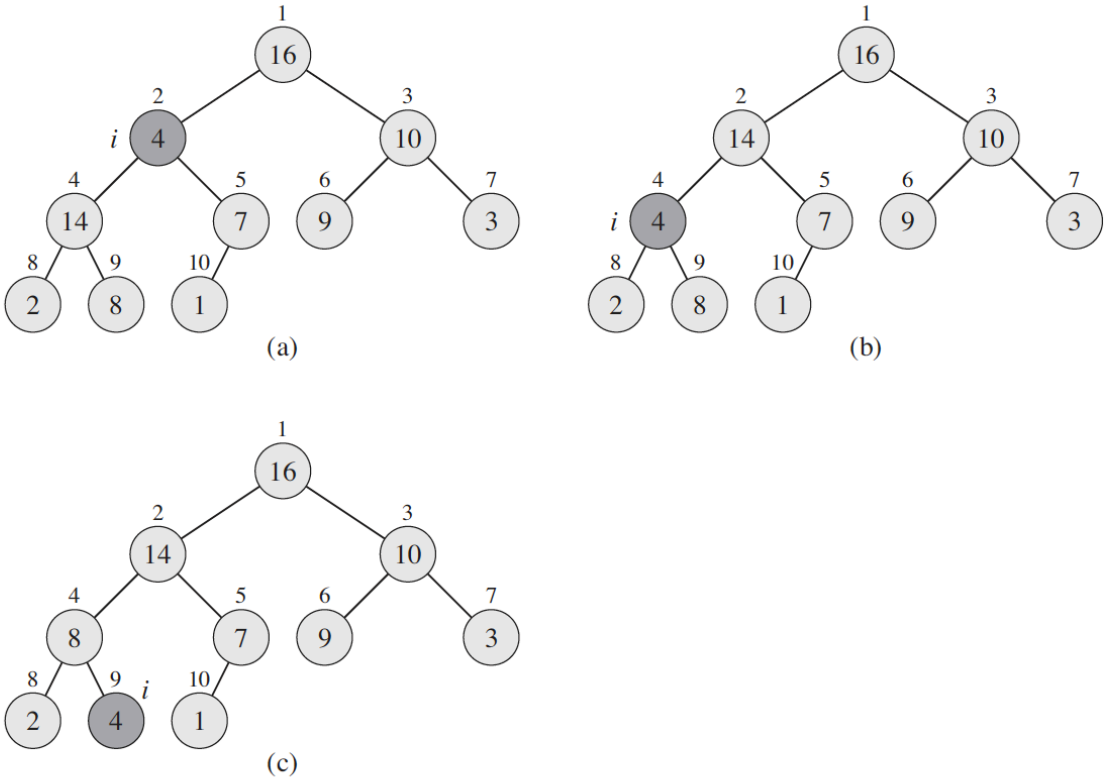


Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

Pseudocode for MAX-HEAPIFY.


```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

Figure 6.2 illustrates the action of MAX-HEAPIFY. At each step, the largest of the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$ is determined, and its index is stored in *largest*. If $A[i]$ is largest, then the subtree rooted at node i is already a max-heap and the procedure terminates. Otherwise, one of the two children has the largest element, and $A[i]$ is swapped with $A[largest]$, which causes node i and its children to satisfy the max-heap property. The node indexed by *largest*, however, now has the original value $A[i]$, and thus the subtree rooted at *largest* might violate the max-heap property. Consequently, we call MAX-HEAPIFY recursively on that subtree.

Example of a Max Heap Violation

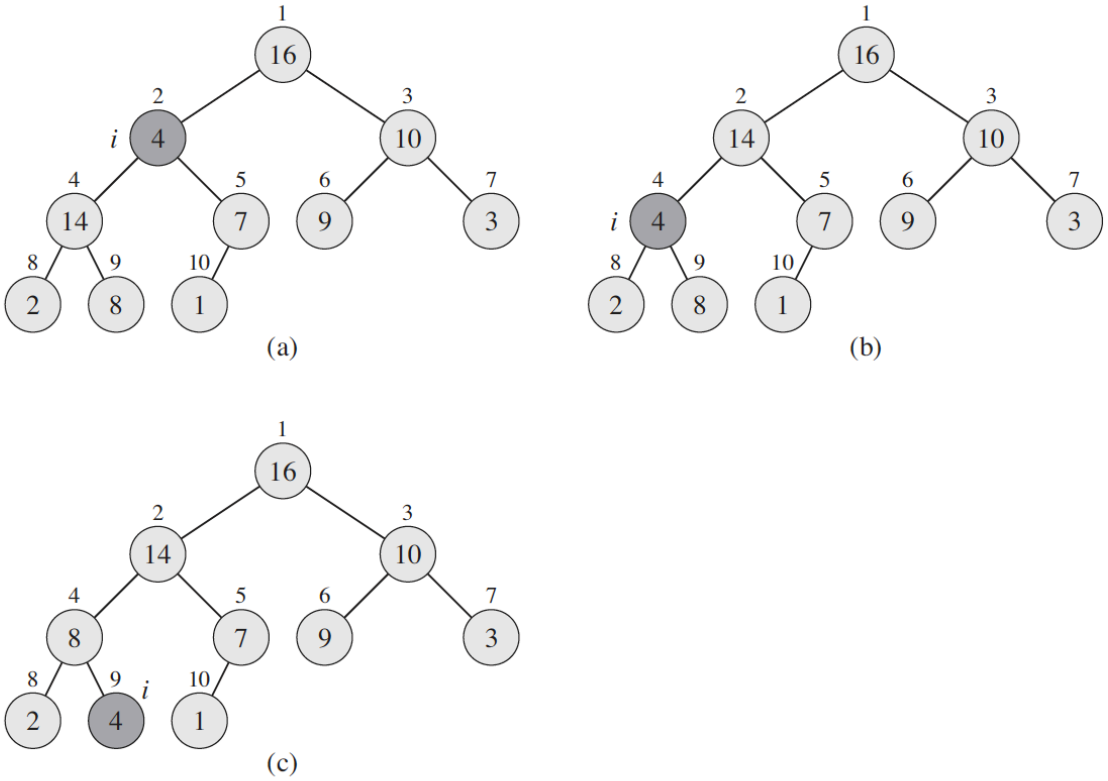


Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

Worse Case for MAX-HEAPIFY (C.L.R.S)

The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$, plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs). The children's subtrees each have size at most $2n/3$—the worst case occurs when the bottom level of the tree is exactly half full—and therefore we can describe the running time of MAX-HEAPIFY by the recurrence

$$T(n) \le T(2n/3) + \Theta(1)$$

Worse Case for MAX-HEAPIFY (C.L.R.S)

The solution to this recurrence, by case 2 of the master theorem,
(solve it yourself and observe the following result).

$$T(n) = O(\log n)$$

$T(n) = T(2n/3) + 1,$

in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

Alternatively we can write the running time of Max Heapify on a node of height h as O(h)

BUILD_MAX_HEAP(A)

BUILD_MAX_HEAP(A)

- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array [1…n], where n=A.length into a max heap

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Figure 6.3 shows an example of the action of BUILD-MAX-HEAP.

Operation BUILD_MAX_HEAP(A)

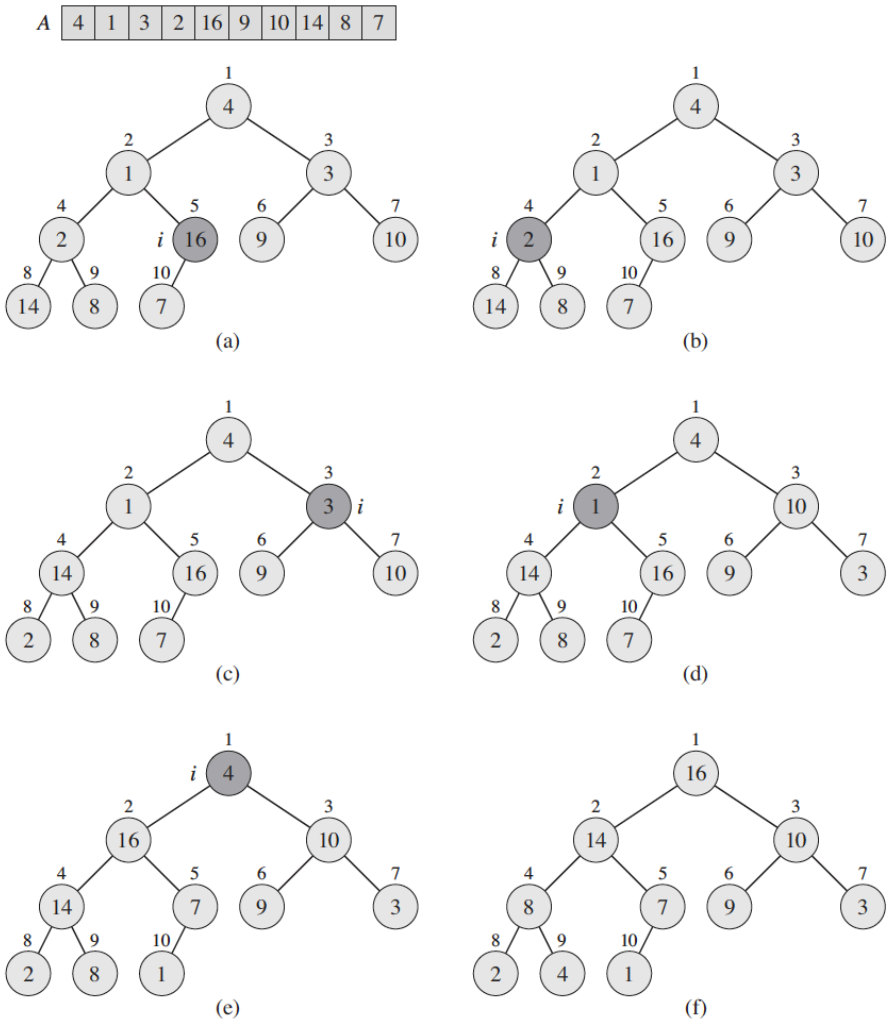


Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

Heap Sort

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Figure 6.4 shows an example of the operation of HEAPSORT after line 1 has built the initial max-heap. The figure shows the max-heap before the first iteration of the **for** loop of lines 2–5 and after each iteration.

The HEAPSORT procedure takes time $O(n \lg n)$, since the call to BUILD-MAX-HEAP takes time $O(n)$ and each of the $n - 1$ calls to MAX-HEAPIFY takes time $O(\lg n)$.

Heap Sort (C.L.R.S)

Operation Heap Sort

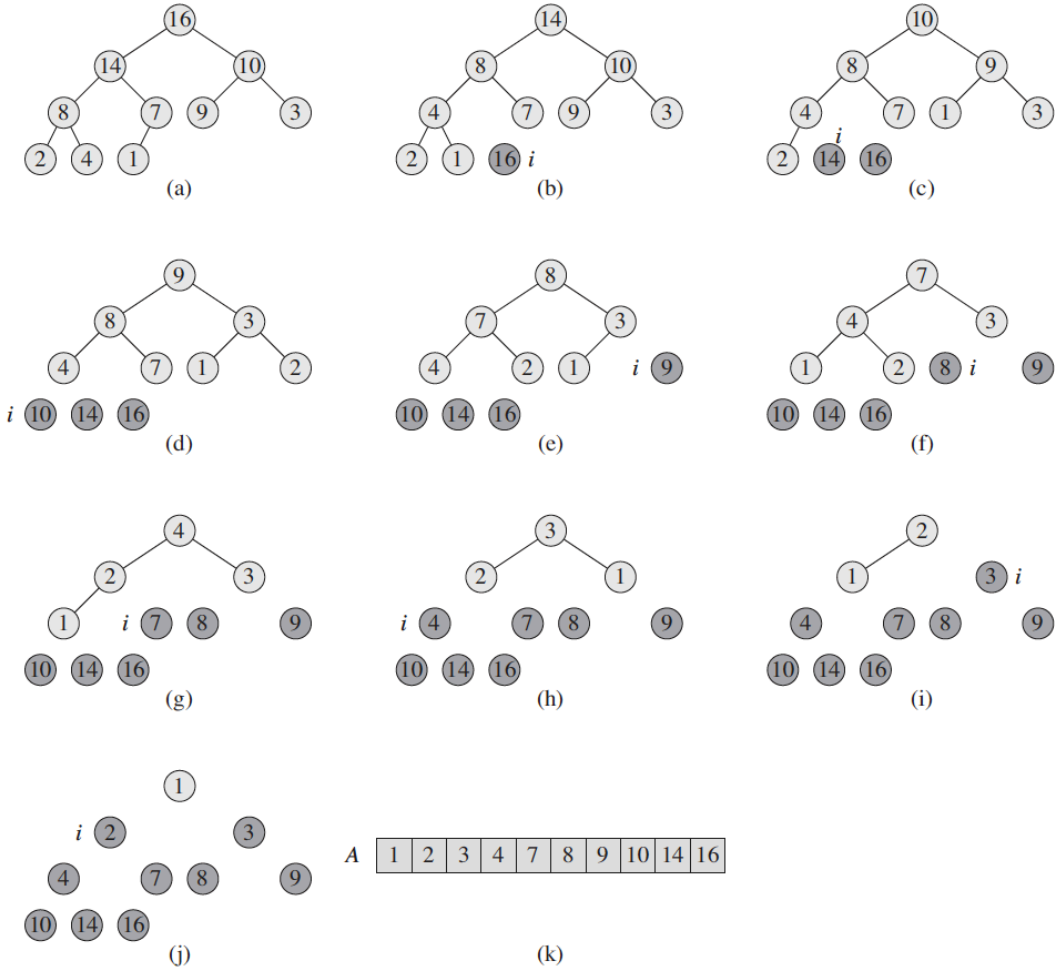


Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

Heap Sort

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Figure 6.4 shows an example of the operation of HEAPSORT after line 1 has built the initial max-heap. The figure shows the max-heap before the first iteration of the **for** loop of lines 2–5 and after each iteration.

The HEAPSORT procedure takes time $O(n \lg n)$, since the call to BUILD-MAX-HEAP takes time $O(n)$ and each of the $n - 1$ calls to MAX-HEAPIFY takes time $O(\lg n)$.

8.4.6

Run-time Summary

The following table summarizes the run-times of heap sort

Case	Run Time	Comments
Worst	$\Theta(n \log(n))$	No worst case
Average	$\Theta(n \log(n))$	
Best	$\Theta(n)$	All or most entries are the same

Summary

Heap Sort is in-place and it is a $\Theta(n \log(n))$ sorting algorithm:

- Convert the unsorted list into a max-heap as complete array
- Pop the top n times and place that object into the vacancy at the end
- It requires $\Theta(1)$ additional memory—it is truly in-place

It is a nice algorithm; however, we will see two other faster $n \ln(n)$ algorithms; however:

- Merge sort requires $\Theta(n)$ additional memory
- Quick sort requires $\Theta(\ln(n))$ additional memory

References

Wikipedia, http://en.wikipedia.org/wiki/Heapsort

[1] Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison Wesley, 1998, §5.2.3, p.144-8.

[2] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, Ch. 7, p.140-9.

[3] Douglas W. Harder, Waterloo University