ECE430.217 Data Structures

Heap sort

Textbook: Weiss Chapter 7.5, Chapter 7.8

Byoungyoung Lee

https://compsec.snu.ac.kr

byoungyoung@snu.ac.kr

Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

Outline

This topic covers the simplest $\Theta(n \ln(n))$ sorting algorithm: *heap sort*

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

Bonus: in-place sorting

Heap Sort

- Let's use min-heap to implement sorting
  - Inserting n objects into a min-heap first
  - Then popping n objects will pop in sorted order

- Heap sort strategy
  - Given an unsorted list with n objects, place them into a heap, and take them out

Run time Analysis of Heap Sort

Taking an object out of a heap with n items requires $\mathbf{O}(\ln(n))$ time

Therefore, taking n objects out requires

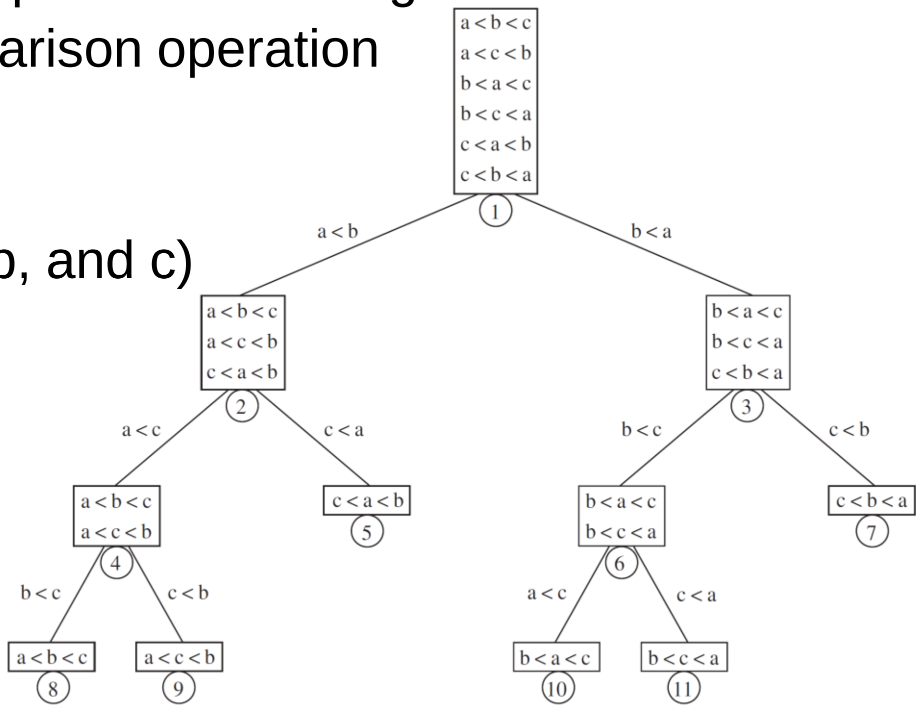
$$\sum_{k=1}^{n} \ln(k) = \ln\left(\prod_{k=1}^{n} k\right) = \ln(n!)$$

Q. What is the asymptotic bound of $\ln(n!)$?

Let’s step back, and think about the runtime of generic sorting algorithms

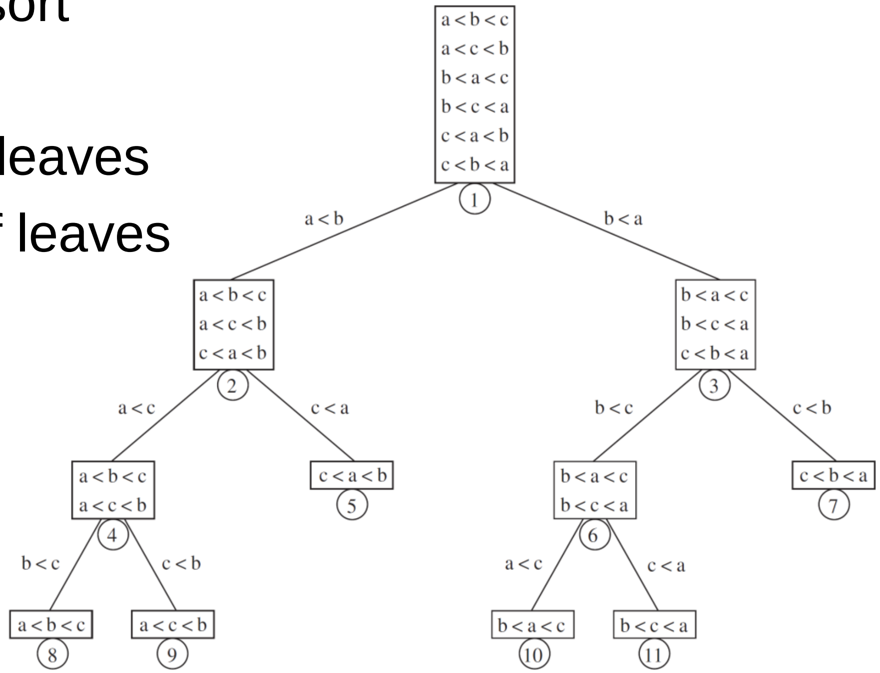
General Lower Bound of Sorting

- **A decision tree**
  - **Abstract representation of any comparison-based sorting algorithm**
  - A decision tree is a binary tree
    - Each node represents a set of possible ordering
    - Each edge represents a comparison operation
  - The right shows the decision tree
    - for three-element sort (i.e., a, b, and c)



General Lower Bound of Sorting

- **A decision tree**
  - Any comparison-based sorting algorithm can be represented with the decision tree
  - The depth of a leaf node
    - The number of comparisons to sort
  - Worst case: the maximum depth of leaves
  - Average case: the average depth of leaves



General Lower Bound of Sorting

- **Lemma (7.1, Weiss p.324)**
  - Let T be a binary tree of depth d. Then T has at most 2^d leaves

- **Proof by induction**
  - **Base case**
    - If d = 0, there's at most one leaf, which is true
  - **Inductive step**
    - Suppose T has two subtrees and the depth of T is d
    - T must have two subtrees, where the depth of each subtree is at most d-1
    - Assuming the induction hypothesis is true for d-1, each subtree has at most 2^{d-1} leaves
    - T has at most 2^d leaves, which proves the lemma.

General Lower Bound of Sorting

- **Lemma (7.2)**
  - A binary tree with L leaves must have depth at least $\lceil \log L \rceil$
  - **Proof**
    - Easy to proof based on Lemma 7.1

- **Theorem (7.6)**
  - Any sorting algorithm that uses only comparisons requires at least $\lceil \log(N!) \rceil$ comparisons in the worst case
  - **Proof:** A decision tree to sort N elements must have N! leaves
    - **Q. why does the decision tree have N! leaves?**

General Lower Bound of Sorting

- **Theorem (7.7)**
  - Any sorting algorithm that uses only comparisons requires $\lceil NlogN \rceil$ comparisons
  - **Proof**
    - $Log(N!) = \log\{N(N-1)(N-2)\cdots(1)\}$

$$
\begin{aligned}
&= \log N + \log(N-1) + \log(N-2) + \cdots + \log 1 \\
&\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log(\tfrac{N}{2}) \\
&\geq \tfrac{N}{2}\log(\tfrac{N}{2}) \\
&\geq \tfrac{N}{2}\log N - \tfrac{N}{2} \\
&= \Omega(N\log N)
\end{aligned}
$$

Recall: $f(n) = \mathbf{O}(g(n)) \Leftrightarrow g(n) = \mathbf{\Omega}(f(n))$

HeapSort: In-place Implementation

Problem:

- The original heap sort solution was out-of-place sorting
- It requires additional memory, a min-heap of size n
- This requires $\Theta(n)$ memory and is therefore not in place

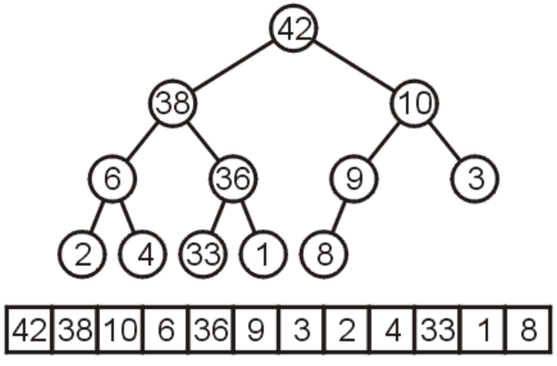
Is it possible to perform a heap sort in-place?

- using at most $\Theta(1)$ memory (a few extra variables)

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

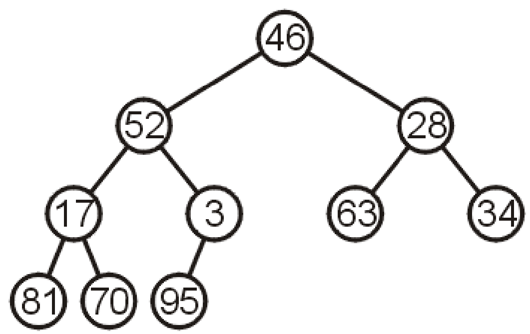


In-place Heapification

Now, consider this unsorted array:

46	52	28	17	3	63	34	81	70	95

This array represents the following complete tree:



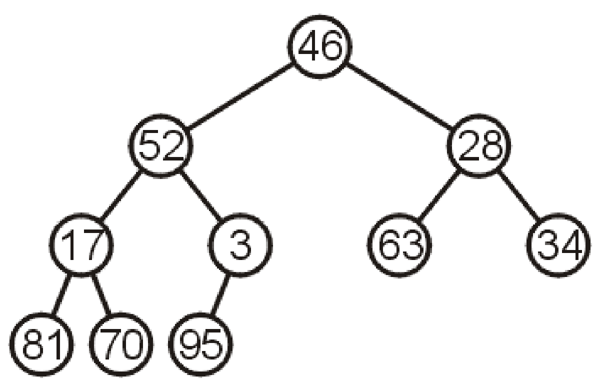
This is neither a min-heap, max-heap, or binary search tree

Goal: In-place conversion of this complete tree into a max heap

In-place Heapification: Two strategies

Two strategies:

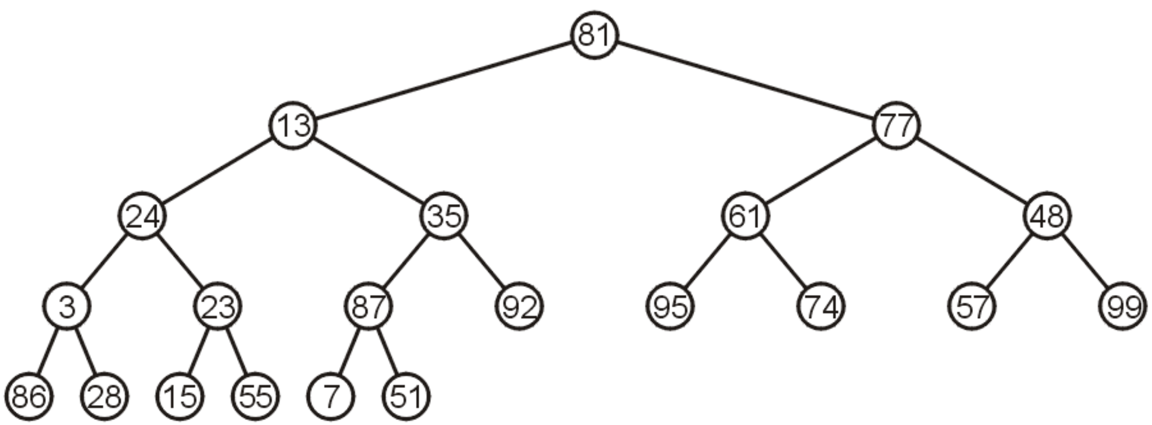
- **Top-down**
  - Assume 46 alone is a max-heap
  - Keep inserting the next element into the existing heap
  - Similar to the strategy for insertion sort
- **Bottom-up**
  - Assume all leaf nodes are already max heaps
  - Make corrections so that previous nodes also form max heaps



In-place Heapification: Bottom-up

Let’s work bottom-up: each leaf node is a max heap on its own

81	13	77	24	35	61	48	3	23	87	92	95	74	57	99	86	28	15	55	7	51

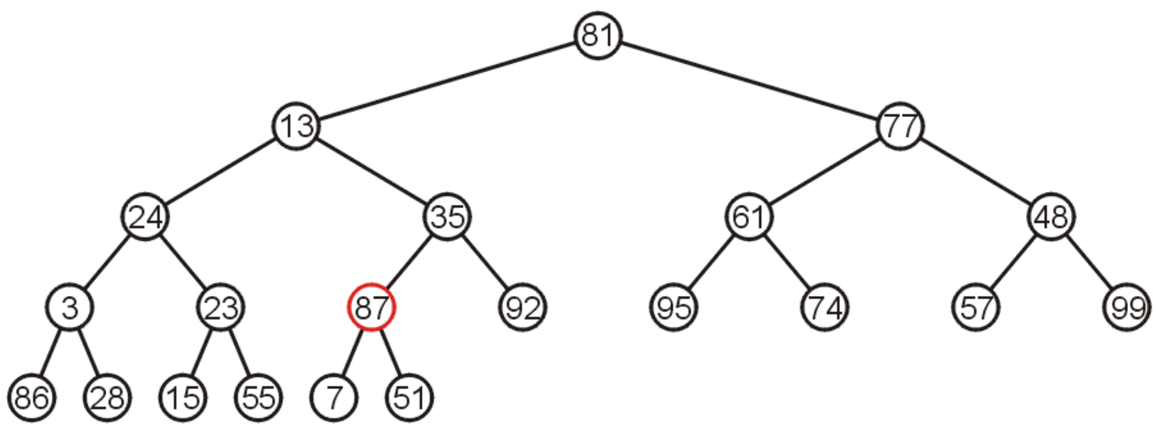


In-place Heapification: Bottom-up

Each leaf node alone is a trivial max-heap, so we don't do any.

Then we move up one level.

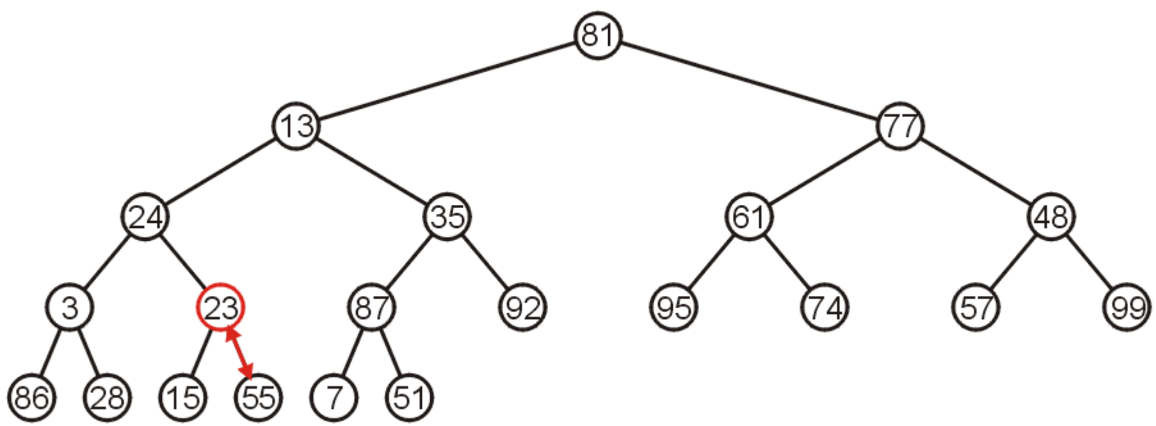
The subtree with 87 as the root is a max-heap



In-place Heapification: Bottom-up

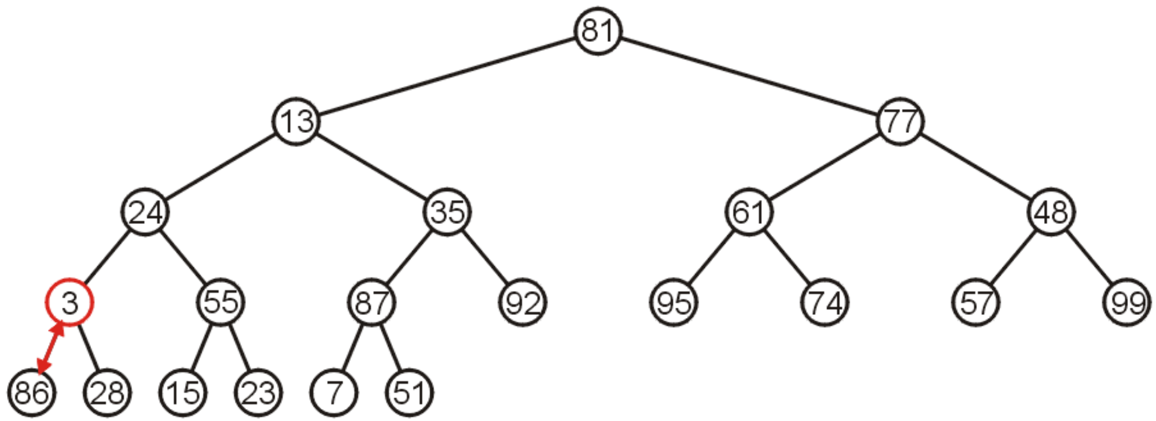
The subtree with 23 is not a max-heap, but swapping it with 55 creates a max-heap

This process is termed **percolating down**



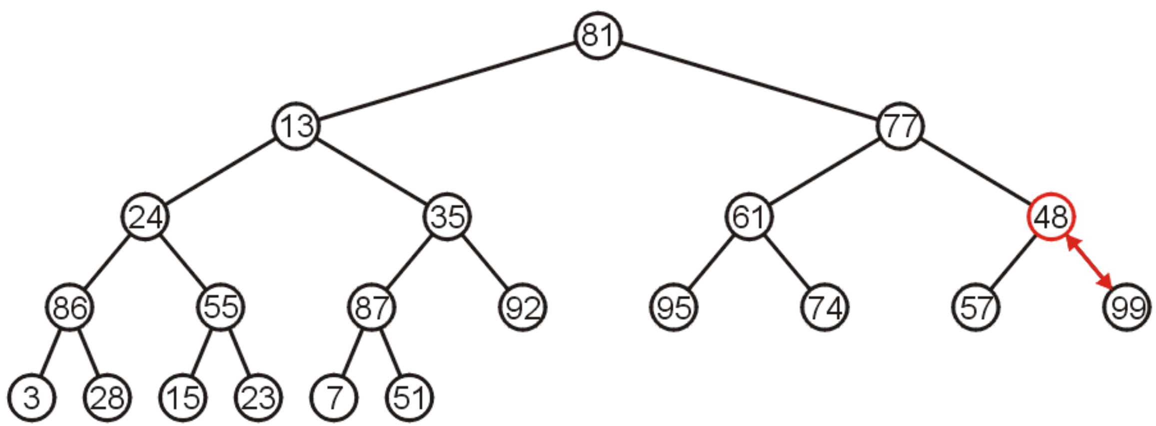
In-place Heapification: Bottom-up

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



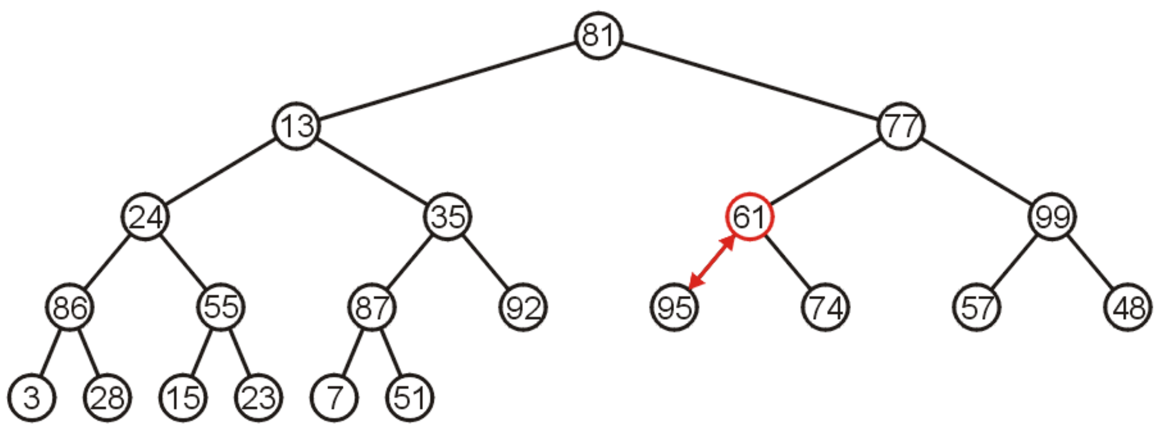
In-place Heapification: Bottom-up

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



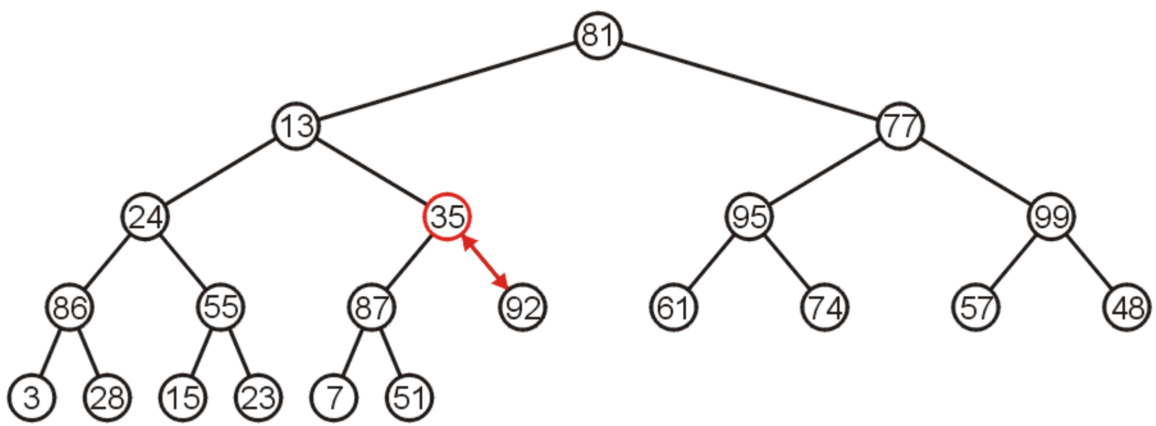
In-place Heapification: Bottom-up

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



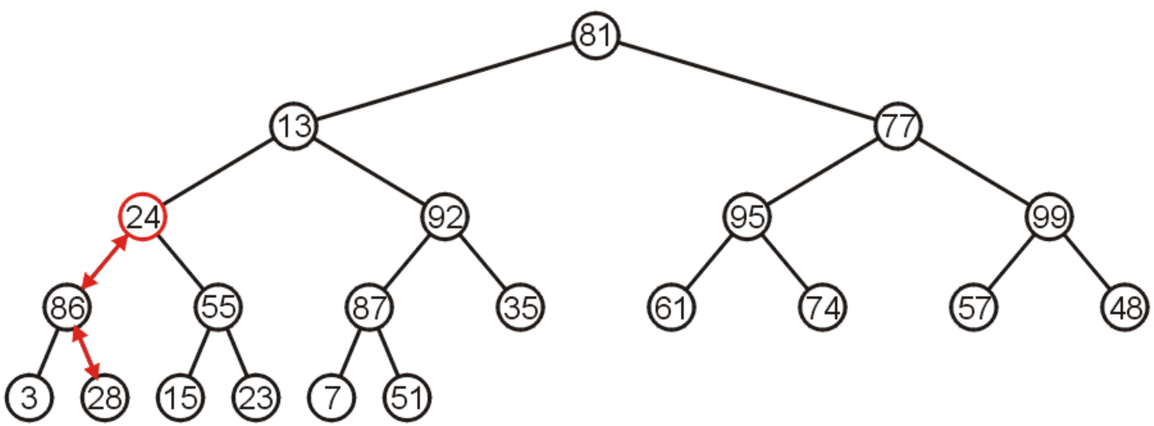
In-place Heapification: Bottom-up

As does swapping 35 and 92



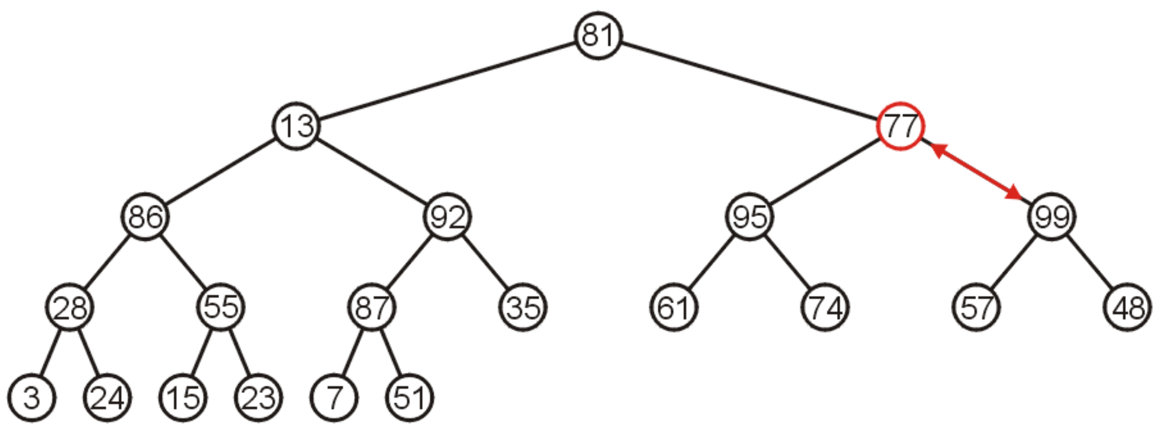
In-place Heapification: Bottom-up

The subtree with root 24 may be converted into a max-heap by first swapping 24 and 86 and then swapping 24 and 28



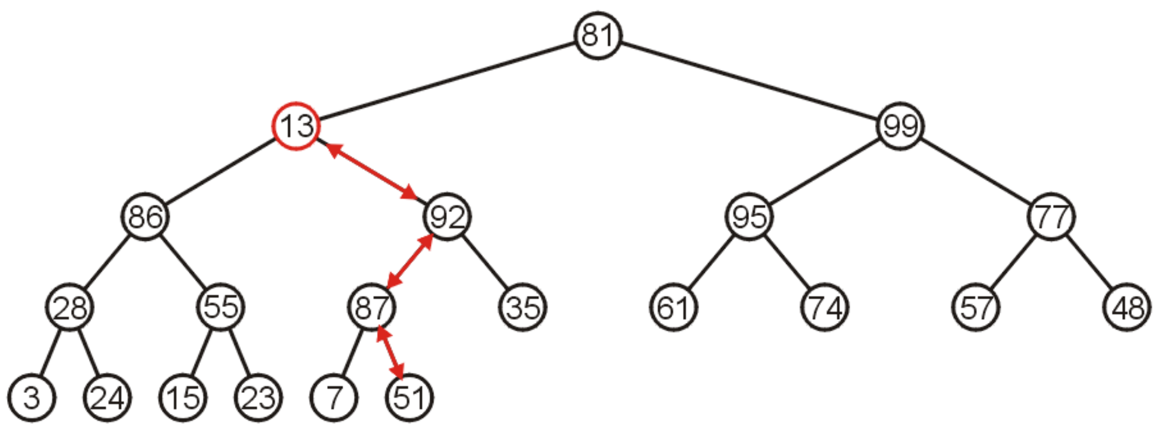
In-place Heapification: Bottom-up

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



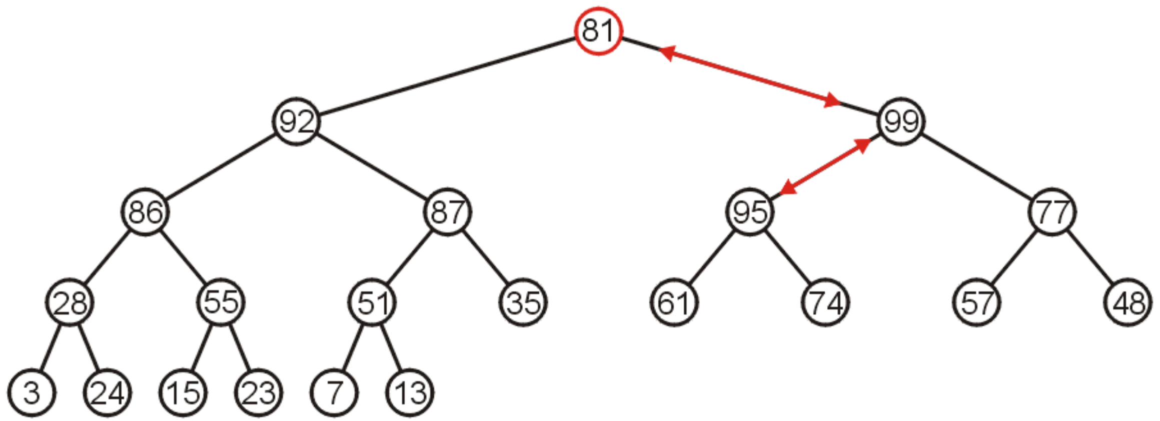
In-place Heapification: Bottom-up

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



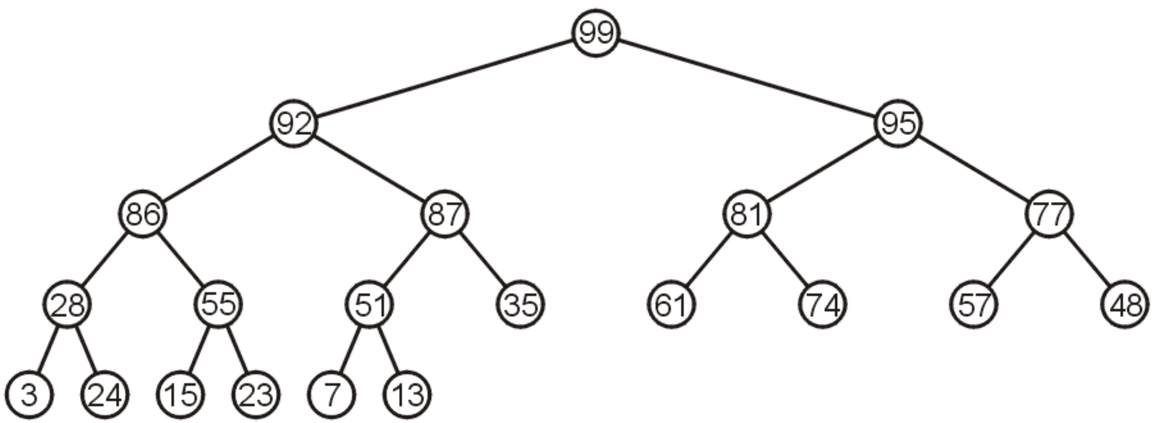
In-place Heapification: Bottom-up

The root need only be percolated down by two levels



In-place Heapification: Bottom-up

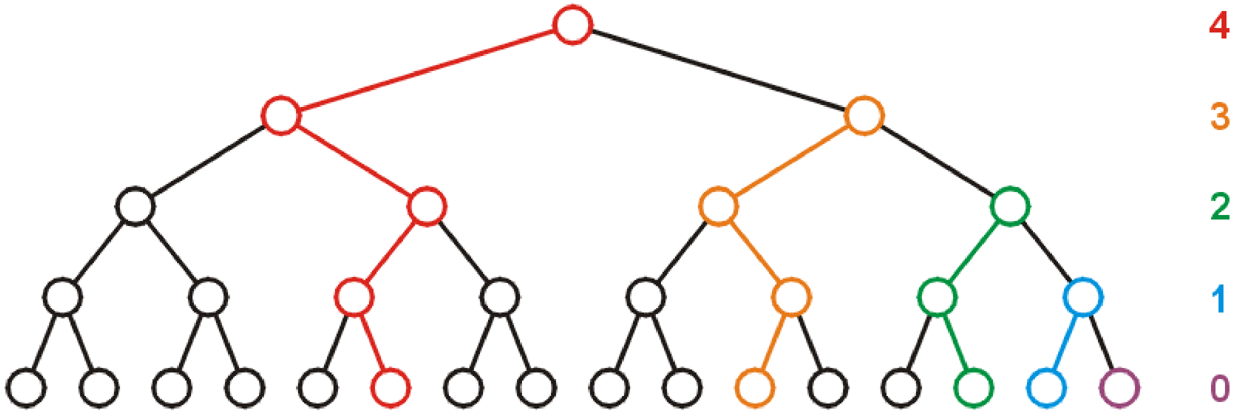
The final product is a max-heap



Run-time Analysis: Bottom-up Heapify

Considering a perfect tree of height h:

- The maximum number of swaps which a second-lowest level would experience is 1, the next higher level, 2, and so on



Run-time Analysis: Bottom-up Heapify

- At depth k
  - There are at most 2^k nodes
  - Nodes at depth k percolate down at most $h - k$ levels
  - In the worst case, this requires a total of $2^k(h - k)$ swaps

- Writing this sum mathematically, we get:

$$\sum_{k=0}^{h} 2^k(h-k) = \left(2^{h+1}-1\right) - (h+1)$$

- Recall that for a perfect tree
  - $n = 2^{h+1} - 1$ and $h + 1 = \lg(n + 1)$

$$\sum_{k=0}^{h} 2^k(h-k) = n - \lg(n+1) = \Theta(n)$$

- So bottom-up heapify takes $\Theta(n)$

Runtime Analysis of Heap Sort

1. Heapification runs in $\Theta(n)$

2. Popping n items from a heap of size n, runs in $\Theta(n \ln(n))$ time

So the total algorithm will run in $\Theta(n \ln(n))$ time

Run-time Summary

The following table summarizes the run-times of heap sort

Case	Run Time	Comments
Worst	$\Theta(n \ln(n))$	No worst case
Average	$\Theta(n \ln(n))$	
Best	$\Theta(n \ln(n))$	No best case

Summary

We have seen our first in-place $\Theta(n \ln(n))$ 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 $n \ln(n)$ algorithms;

- 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] Weiss, *Data Structures and Algorithm Analysis in C++, 3rd Ed.*, Addison Wesley, §7.5, p.270-4.