

Heap and Heap sort

a.k.a. *heapsort*

Outline

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

We will:

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

Bonus: may be performed in place

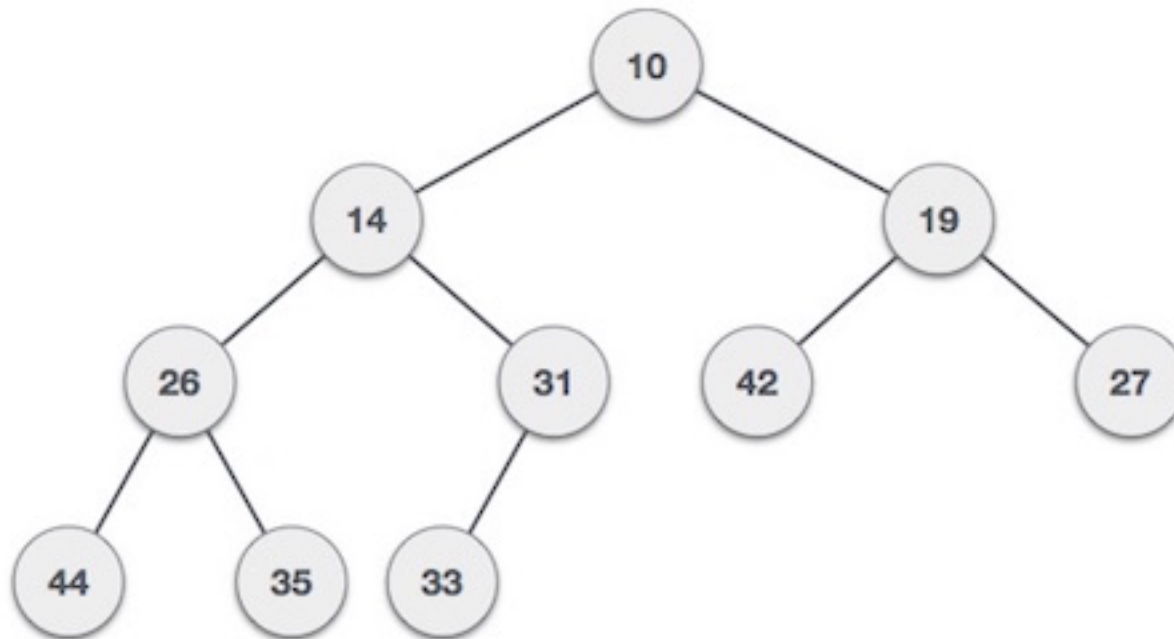
Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then –

$$\text{key}(\alpha) \geq \text{key}(\beta)$$

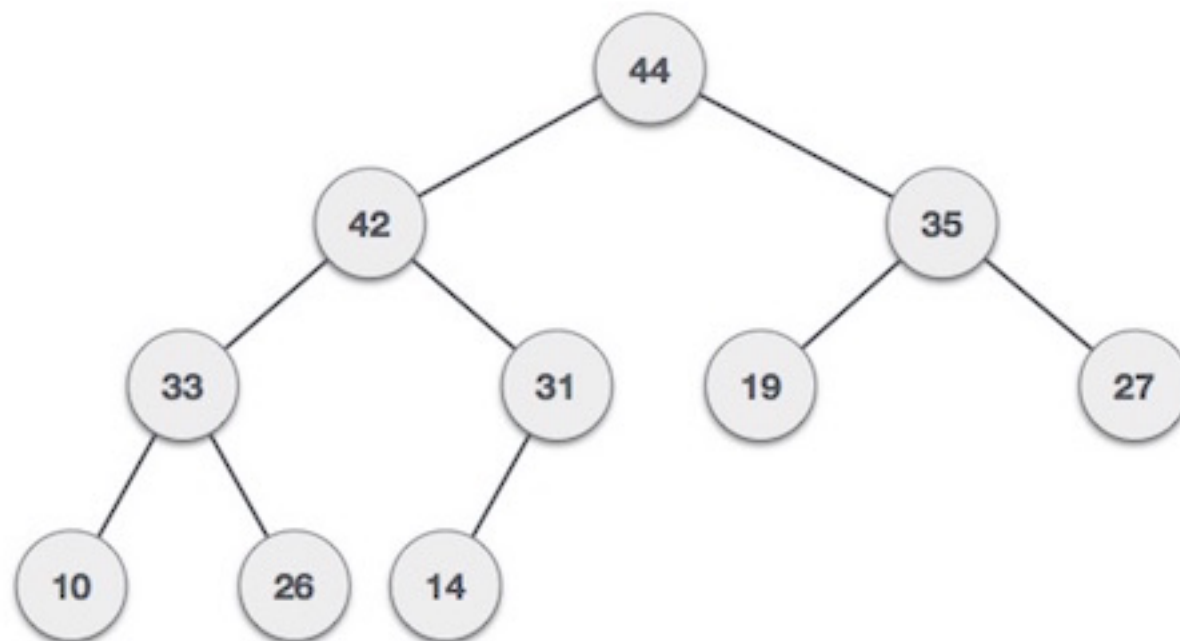
As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types –

For Input → 35 33 42 10 14 19 27 44 26 31

Min-Heap – Where the value of the root node is less than or equal to either of its children.



Max-Heap – Where the value of the root node is greater than or equal to either of its children.



Both trees are constructed using the same input and order of arrival.

Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

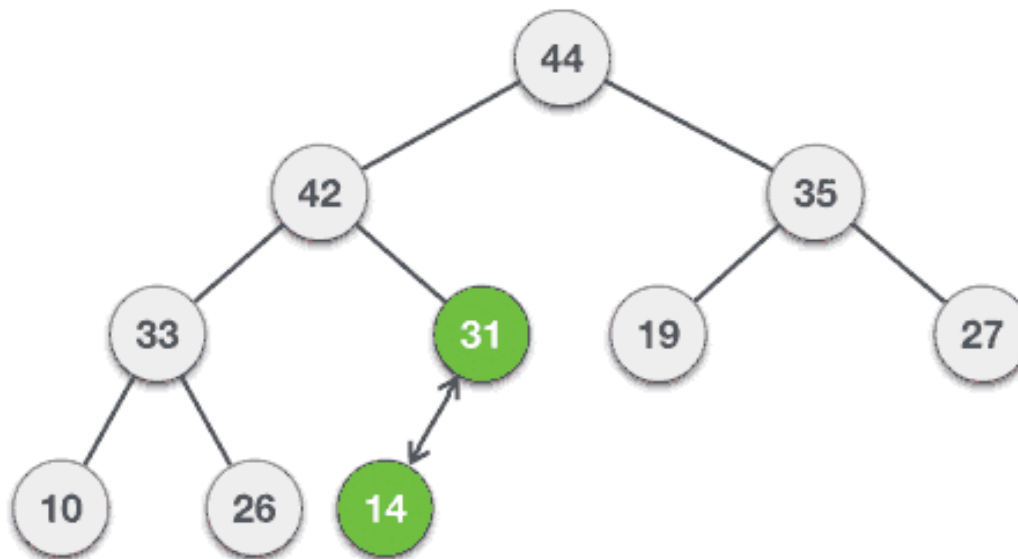
We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

- Step 1** – Create a new node at the end of heap.
- Step 2** – Assign new value to the node.
- Step 3** – Compare the value of this child node with its parent.
- Step 4** – If value of parent is less than child, then swap them.
- Step 5** – Repeat step 3 & 4 until Heap property holds.

Note – In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Example of building a Max Heap:

Input: 35 33 42 10 14 19 27 44 26 31



Heap Sort

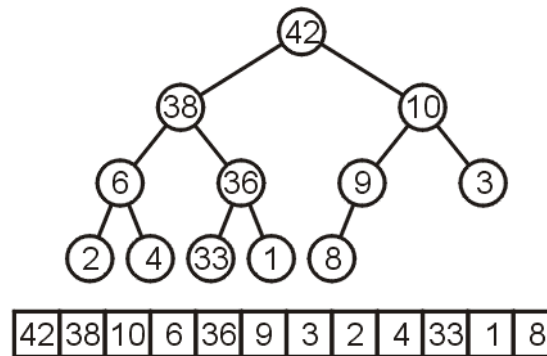
Recall that inserting n objects into a min-heap and then taking n objects will result in them coming out in order

Strategy: given an unsorted list with n objects, place them into a heap, and take them out

In-place Implementation

Implementing a max-heap:

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

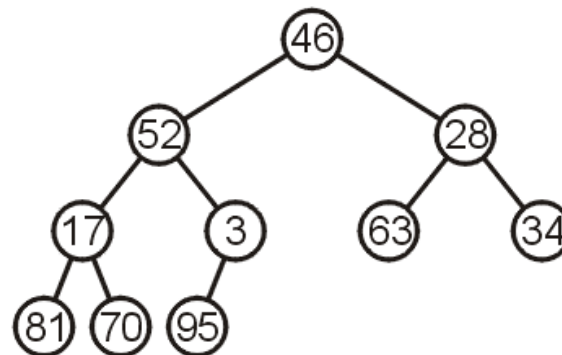


In-place Heapification

Now, consider this unsorted array:

| | | | | | | | | | |
|----|----|----|----|---|----|----|----|----|----|
| 46 | 52 | 28 | 17 | 3 | 63 | 34 | 81 | 70 | 95 |
|----|----|----|----|---|----|----|----|----|----|

This array represents the following complete tree:



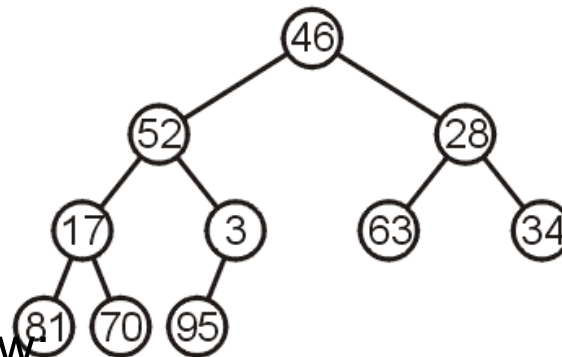
This is neither a min-heap, max-heap, or binary search tree

In-place Heapification

Now, consider this unsorted array:

| | | | | | | | | | |
|----|----|----|----|---|----|----|----|----|----|
| 46 | 52 | 28 | 17 | 3 | 63 | 34 | 81 | 70 | 95 |
|----|----|----|----|---|----|----|----|----|----|

Additionally, because arrays start at 0 (we started at entry 1 for binary heaps), we need different formulas for the children and parent



The formulas are now:

Children

$$2*k + 1 \quad 2*k + 2$$

Parent

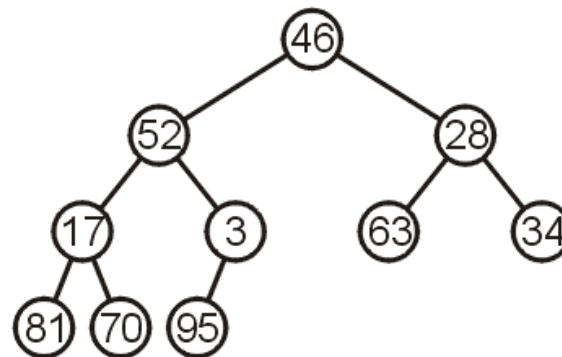
$$(k + 1)/2 - 1$$

In-place Heapification

Can we convert this complete tree into a max heap?

Restriction:

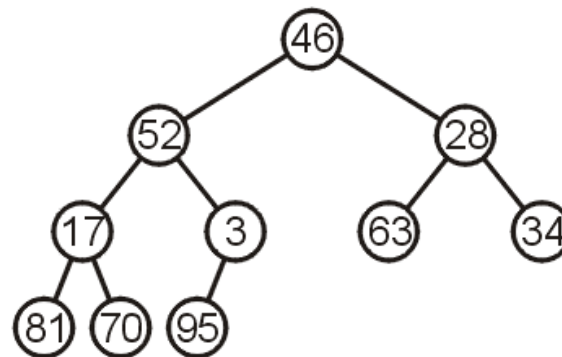
- The operation must be done in-place



In-place Heapification

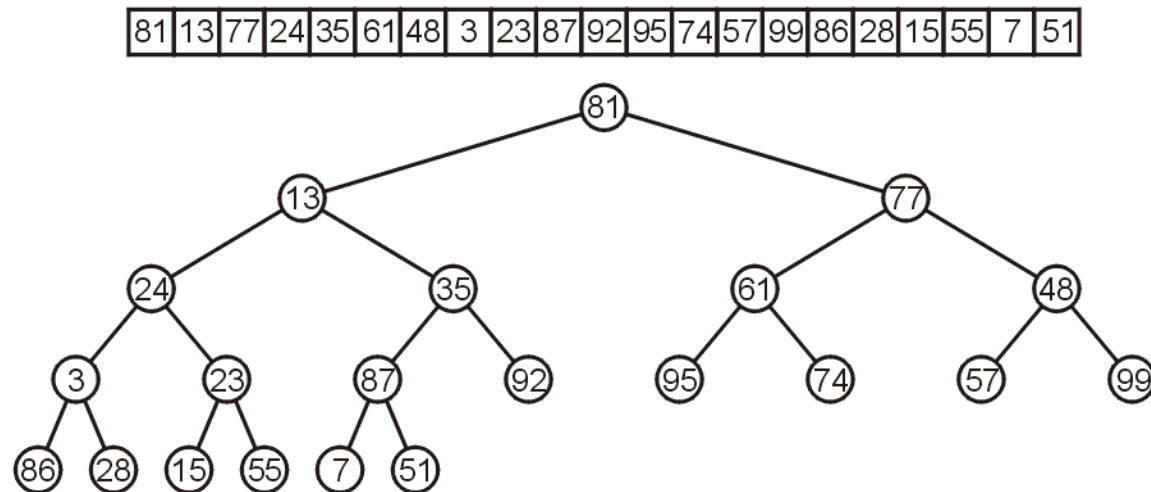
Two strategies:

- Assume 46 is a max-heap and keep inserting the next element into the existing heap (similar to the strategy for insertion sort)
- Start from the back: note that all leaf nodes are already max heaps, and then make corrections so that previous nodes also form max heaps



In-place Heapification

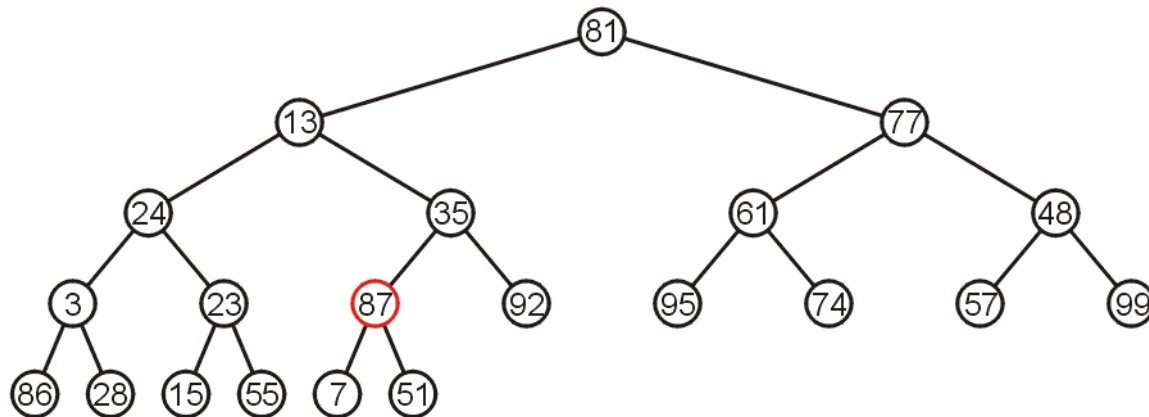
Let's work bottom-up: each leaf node is a max heap on its own



In-place Heapification

Starting at the back, we note that all leaf nodes are trivial heaps

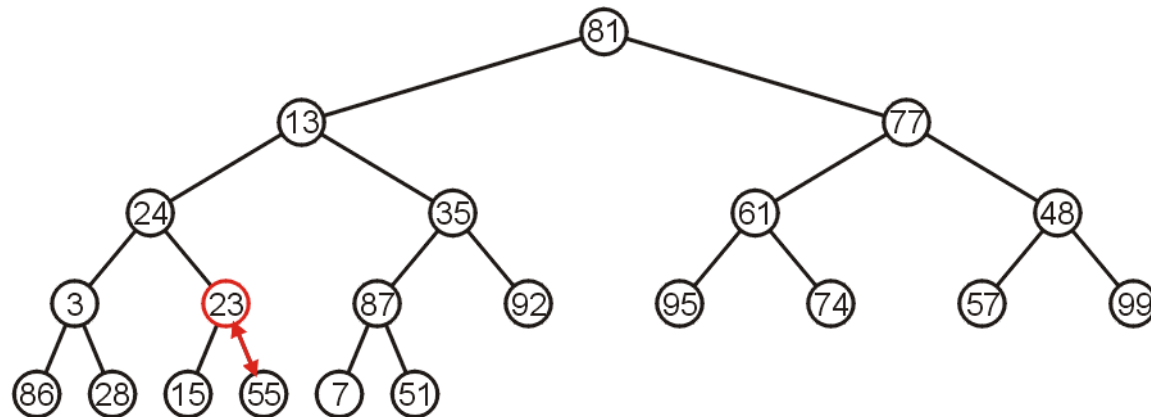
Also, the subtree with 87 as the root is a max-heap



In-place Heapification

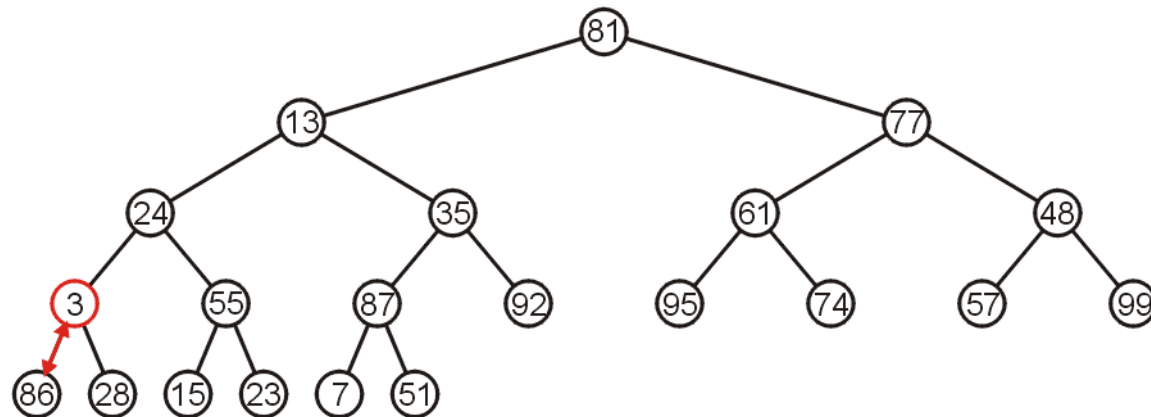
The subtree with 23 is not a max-heap, but swapping it with 55 creates a max-heap

This process is termed *percolating down*



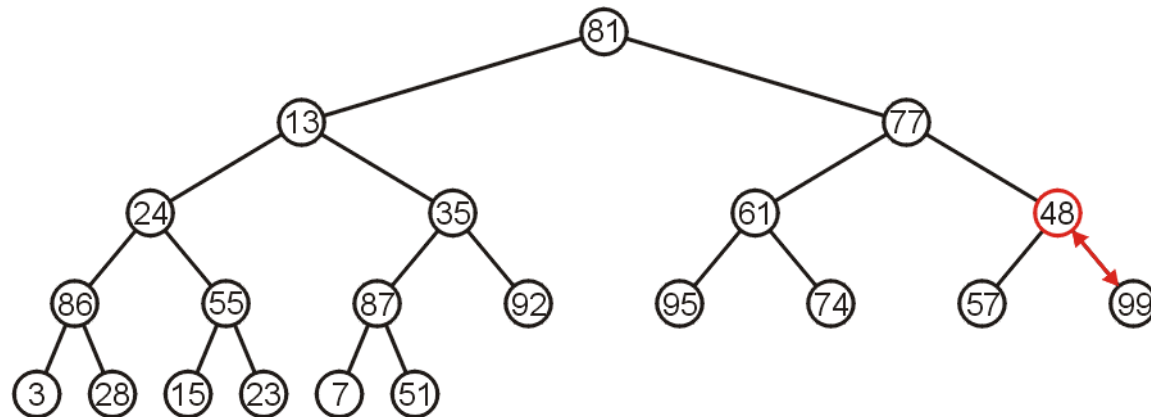
In-place Heapification

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



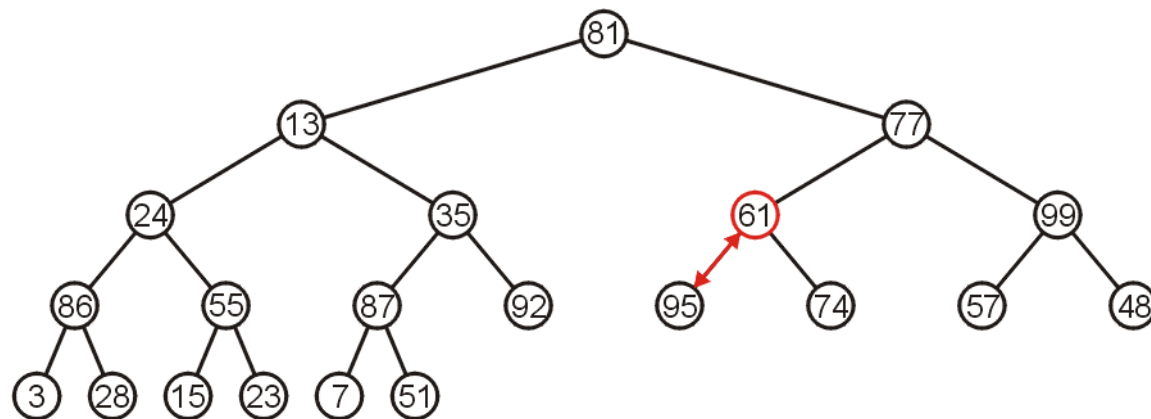
In-place Heapification

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



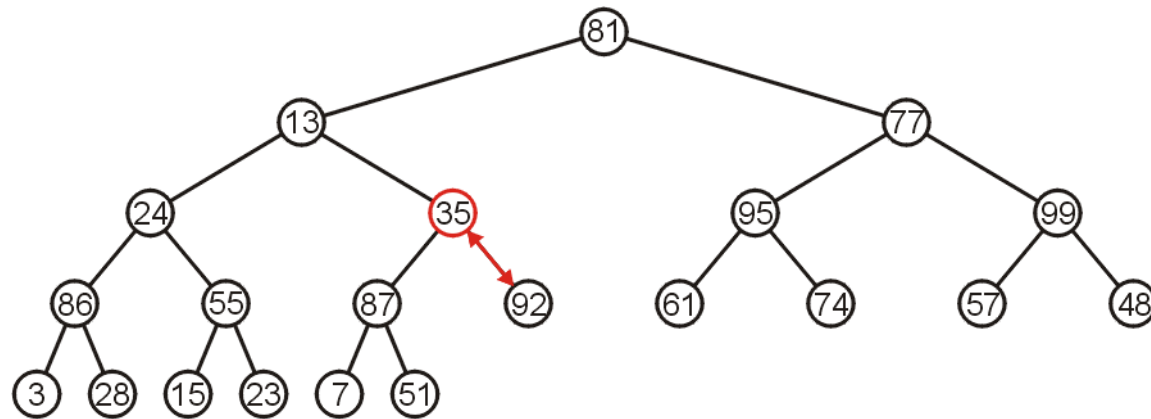
In-place Heapification

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



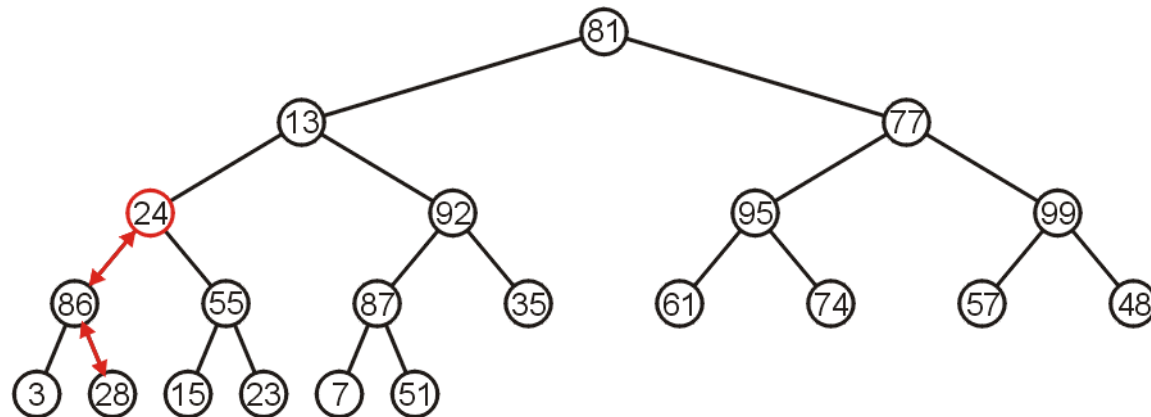
In-place Heapification

As does swapping 35 and 92



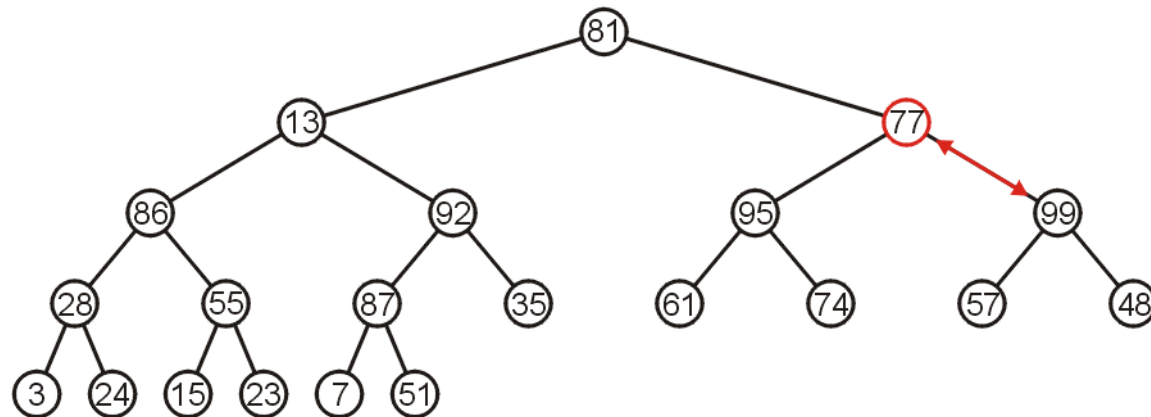
In-place Heapification

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



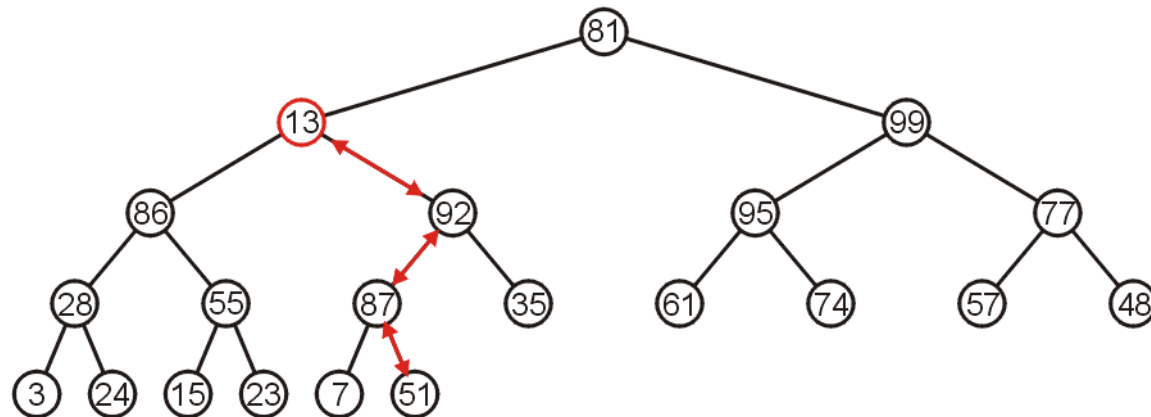
In-place Heapification

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



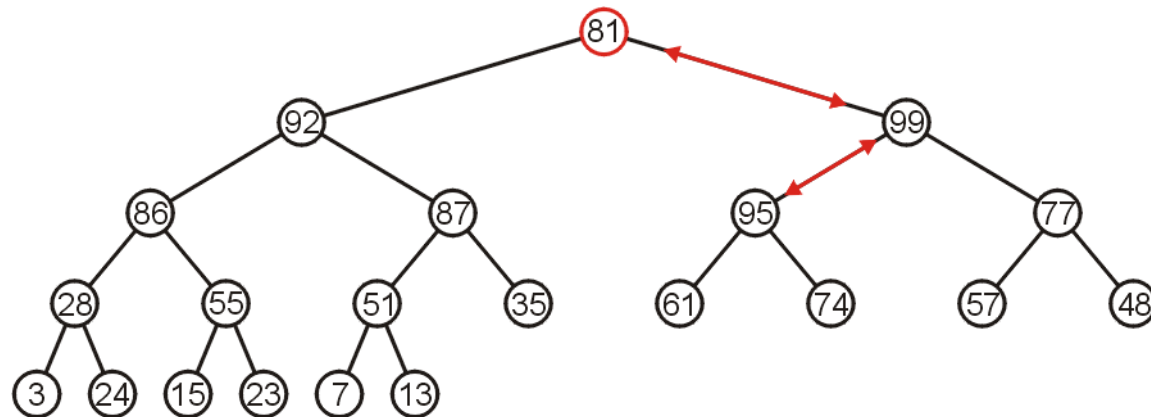
In-place Heapification

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



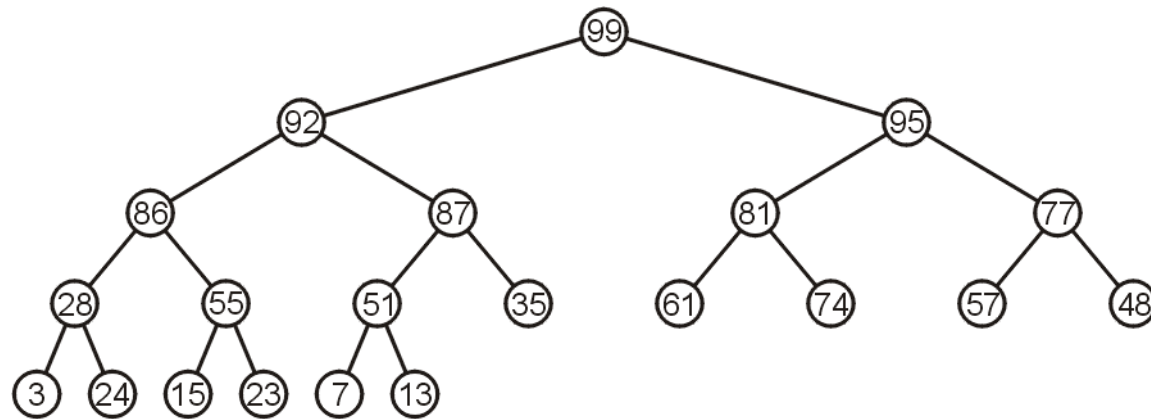
In-place Heapification

The root need only be percolated down by two levels



In-place Heapification

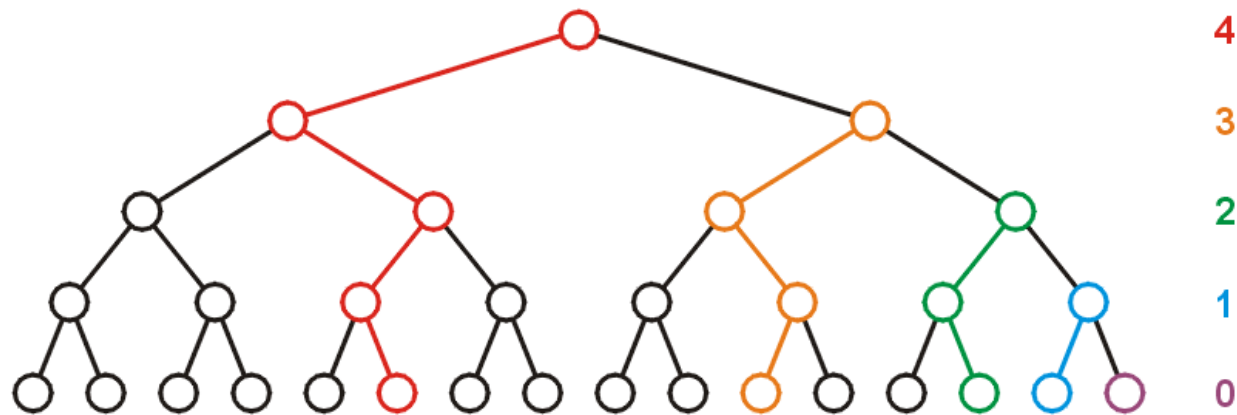
The final product is a max-heap



Run-time Analysis of 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



Example Heap Sort

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 $n/2$

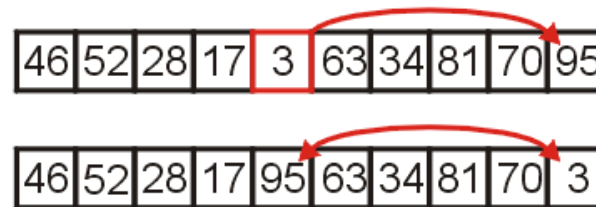
Thus we start with position $10/2 = 5$



This image cannot currently be displayed.

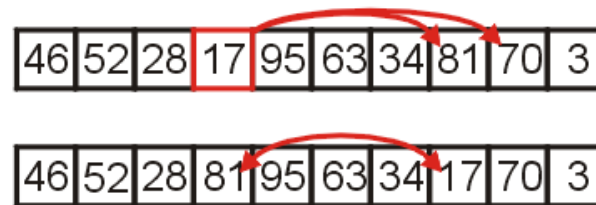
Example Heap Sort

We compare 3 with its child and swap them



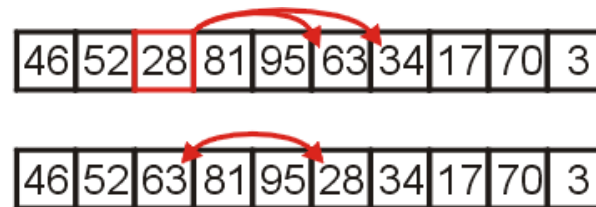
Example Heap Sort

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



Example Heap Sort

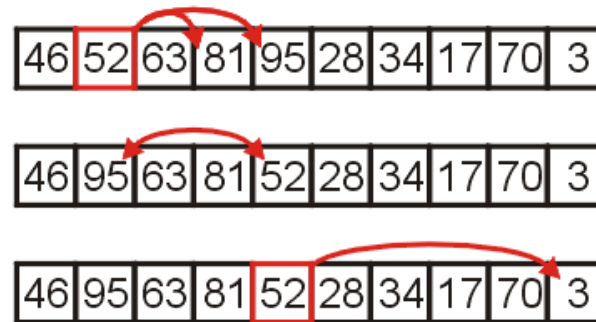
We compare 28 with its two children, 63 and 34, and swap it with the largest child



Example Heap Sort

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

- Recursing, no further swaps are needed



Example Heap Sort

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



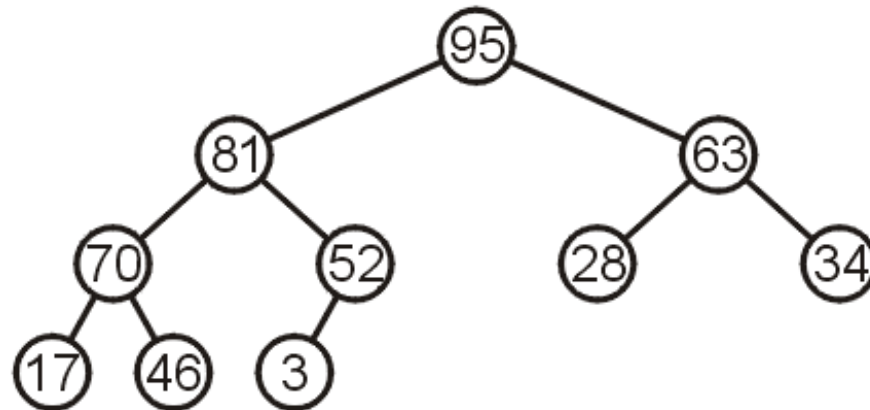
Heap Sort Example

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 |
|----|----|----|----|----|----|----|----|----|---|

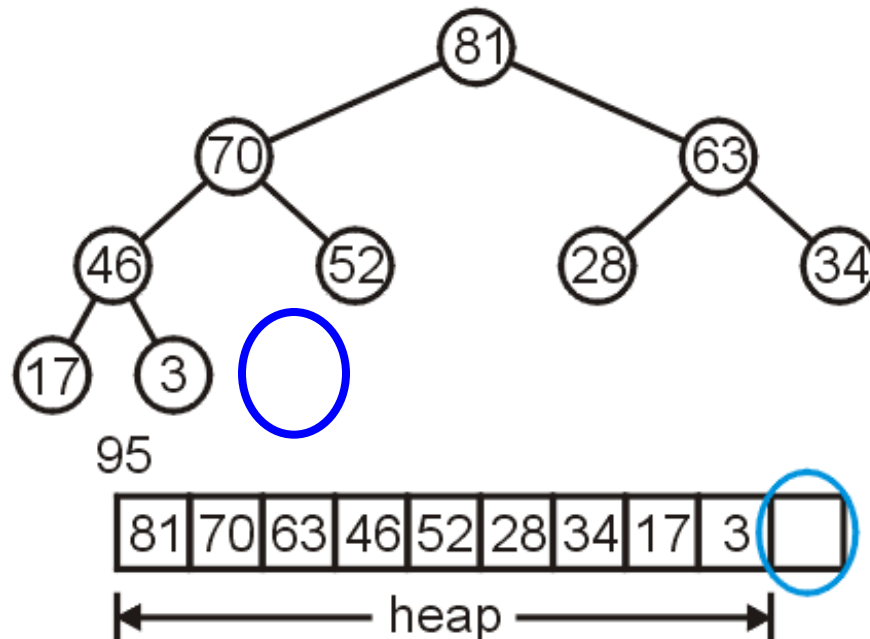


Heap Sort Example

Suppose we pop the maximum element of this heap

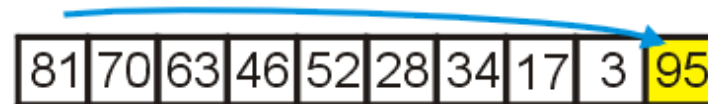


This leaves a gap at the back of the array:

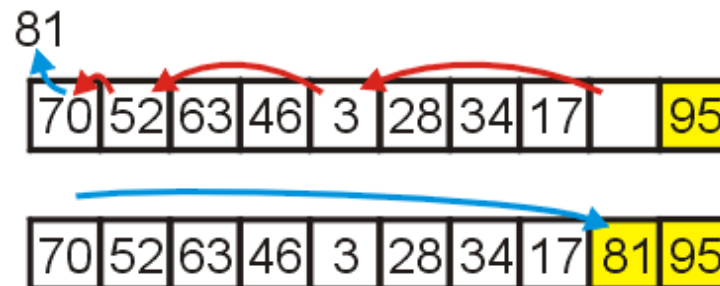


Heap Sort Example

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



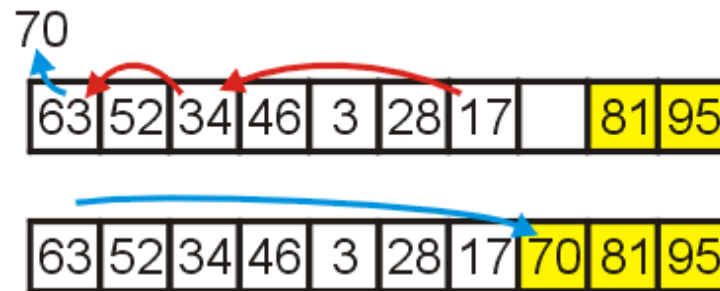
Repeat this process: pop the maximum element, and then insert it at the end of the array:



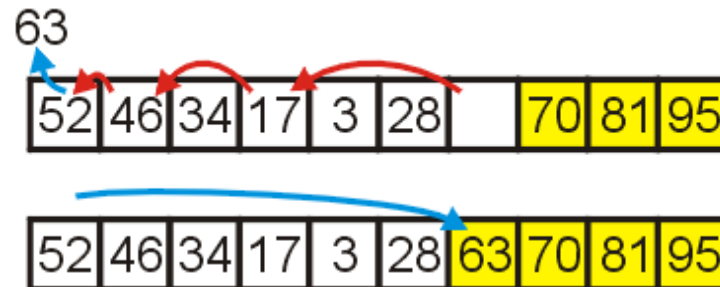
Heap Sort Example

Repeat this process

- Pop and append 70



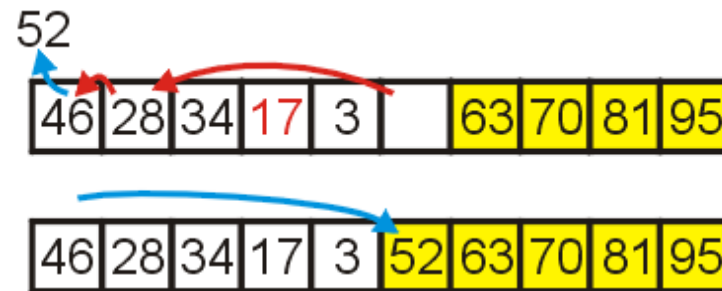
- Pop and append 63



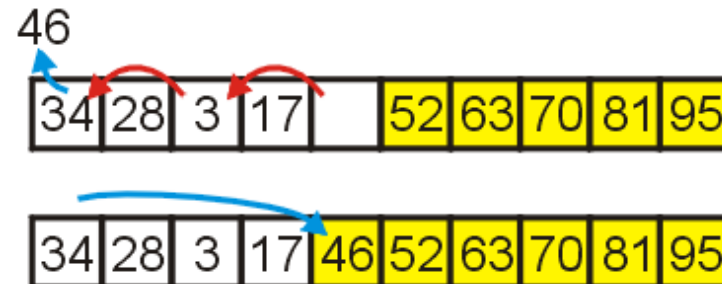
Heap Sort Example

We have the 4 largest elements in order

- Pop and append 52



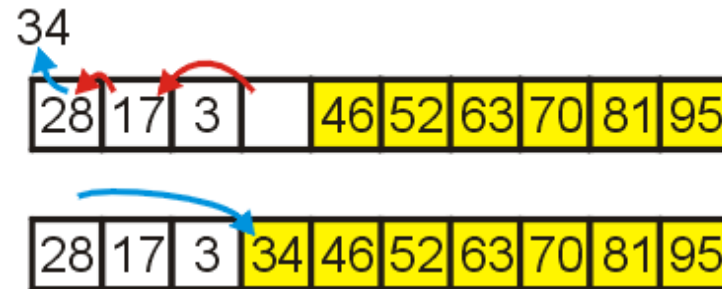
- Pop and append 46



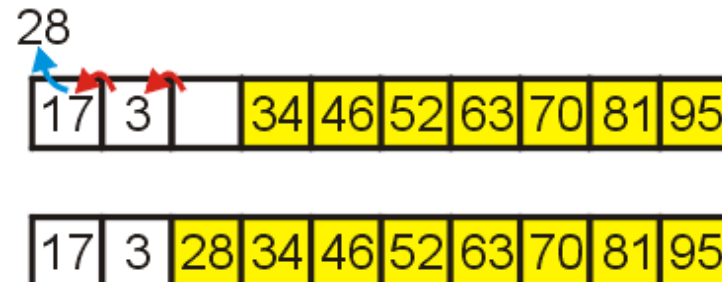
Heap Sort Example

Continuing...

- Pop and append 34

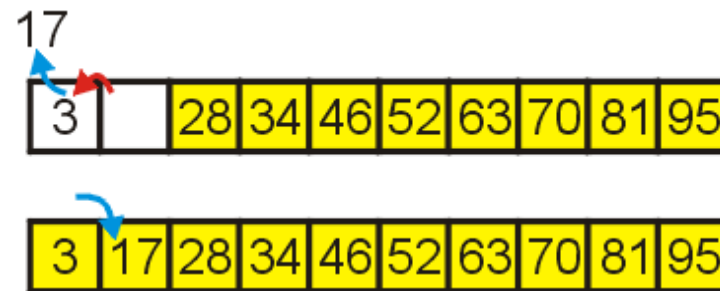


- Pop and append 28



Heap Sort Example

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



Black Board Example

Sort the following 12 entries using heap sort

34, 15, 65, 59, 79, 42, 40, 80, 50, 61, 23, 46

Heap Sort

Heapification runs in $\Theta(n)$

Popping n items from a heap of size n , as we saw, runs in $\Theta(n \ln(n))$ time

- We are only making one additional copy into the blank left at the end of the array

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

Heap Sort

There are no worst-case scenarios for heap sort

- Dequeueing from the heap will always require the same number of operations regardless of the distribution of values in the heap

There is one best case: if all the entries are identical, then the run time is $\Theta(n)$

The original order may speed up the *heapification*, however, this would only speed up an $\Theta(n)$ portion of the algorithm

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 | $\Theta(n)$ | All or most entries are the same |

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

These slides are provided for the ECE 250 *Algorithms and Data Structures* course. The material in it reflects Douglas W. Harder's best judgment in light of the information available to him at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. Douglas W. Harder accepts no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.