ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo
Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 8.4.
Assistances and comments will be acknowledged.

8.4 Heap Sort (*heapsort*)

We will now look at our first $\Theta(n \ln(n))$ algorithm: heap sort. It will use a data structure that we have already seen: a binary heap.

8.4.1 Strategy and Run-time Analysis

Given a list of n objects, insert them into a min-heap and take them out in order. Now, in the worst case, both a push into and a pop from a binary min-heap with k entries is $\Theta(\ln(k))$. Therefore, the total run time will be

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

because the sum of logarithms is the logarithm of the products: $\ln(a) + \ln(b) = \ln(ab)$. The question is, what is the asymptotic growth of $\ln(n!)$? Such a question is best left to Maple:

```
> asympt( ln( n! ), n );
```

$$\left(\ln(n)-1\right)n+\frac{1}{2}\ln(n)+\frac{1}{2}\ln(2\pi)+\frac{1}{2}\ln(n)+\frac{1}{12n}-\frac{1}{360n^3}+\mathrm{O}\left(\frac{1}{n^5}\right)$$

The dominant term in this asymptotic series is $(\ln(n) - 1)n$ and thus the run time is $O(n \ln(n))$. If you plot both $\ln(n!)$ and $n \ln(n)$, you will note that they appear to be grown at approximately the same rate.

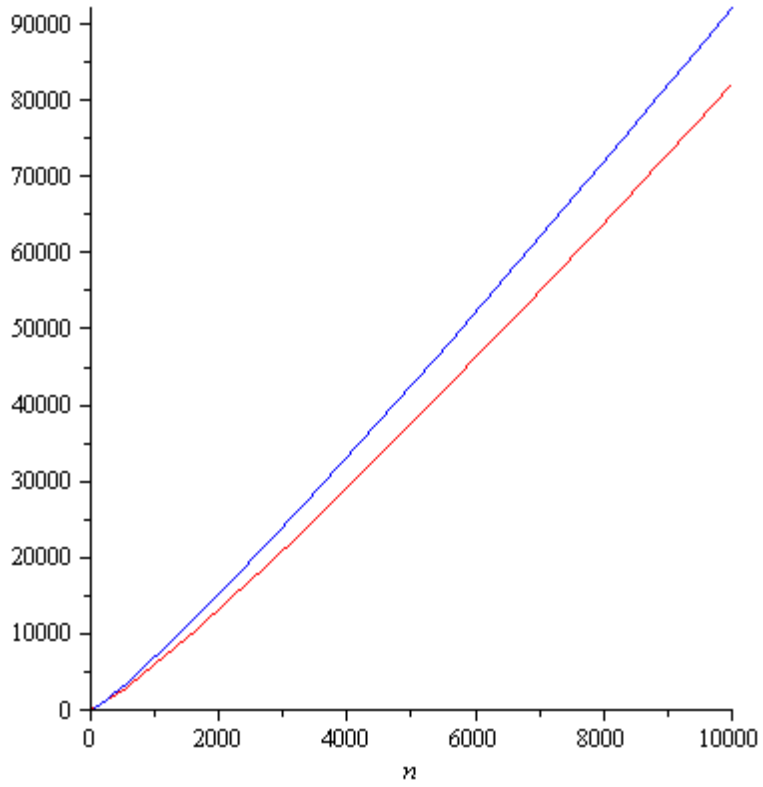


Figure 1. A plot of $\ln(n!)$ in red and $n \ln(n)$ in blue.

Note that it is not possible to calculate ln(200!) directly as $200! \approx 7.89 \times 10^{375}$ which is greater than the largest double-precision floating-point number.

8.4.2 In-place Implementation

Issue: this implementation requires an additional array for the heap... Is it possible to do a heap sort in-place? That is, can we do heap sort with only $\Theta(1)$ additional memory?

Suppose we have a max-heap as is shown in Figure 2.

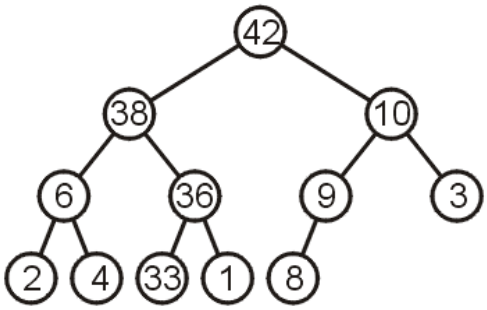


Figure 2. A max-heap.

The array representation of this max-heap is

1	2	3	4	5	6	7	8	9	10	11	12
42	38	10	6	36	9	3	2	4	33	1	8

If we pop the top entry of the heap, this creates a vacancy in the last position:

1	2	3	4	5	6	7	8	9	10	11	12
38	36	10	6	33	9	3	2	4	8	1	**?**

We could place 42 into this last position but ignore it the next time we pop 38:

1	2	3	4	5	6	7	8	9	10	11	12
36	33	10	6	8	9	3	2	4	1	**?**	**42**

After *n* pops, we would have a sorted list.

The problem, however, is that we do not start with a max heap. Consider, for example, the unsorted array

0	1	2	3	4	5	6	7	8	9
46	52	28	17	3	63	34	81	70	95

If we interpret this as a binary tree, we have the tree shown in Figure 3.

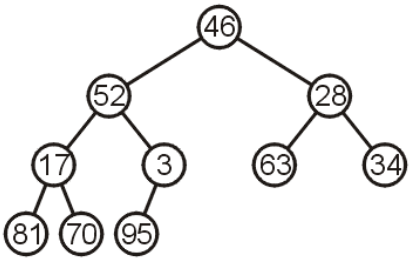


Figure 3. The tree-representation of the given unsorted array.

Unfortunately, this is neither a min-heap, a max-heap, nor a binary search tree. It's just a binary tree. We need to convert this tree into a max-heap and we must do it in place.

Another issue is indexing: recall that with binary heaps, we left the location 0 empty so that we could use the simple formulas that for the entry `k`, its children are locations `2*k` and `2*k + 1` and its parent is in location `k/2`. Because all arrays start at 0, we can use the formulas `2*k + 1`, `2*k + 2` for the children and `(k + 1)/2 - 1` for the parent. Fortunately, we will have to find the parent of an entry at most n times.

8.4.3 In-place Heapification

In order to convert an arbitrary complete tree into a binary max-heap, there are two strategies we could use:

1. Similar to insertion sort, assume the root is a binary max-heap and keep inserting the next entry in the array into the existing max-heap, or
2. Start from the bottom and note that all the leaf nodes are already valid max-heaps. Now, carry on working from right to left, and for each previous element, make whatever changes are necessary to convert the tree rooted at that node into a max-heap working all the way back to the first entry.

Let us work from the leaves up. In order to do this, let's consider the larger tree shown in Figure 4.

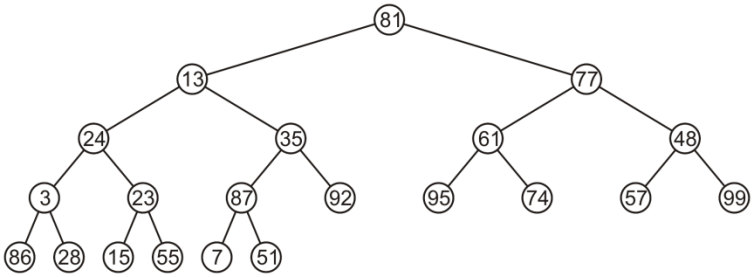


Figure 4. The binary tree representation of an unsorted list with 21 entries.

Starting from the last entry, 51, and working our way in a reverse-breadth-first traversal, we note that all the leaf nodes are trivial max-heaps of size 1. Now, look at node 87: it, too, is a max-heap; however, the two trees rooted at 23 and 3 are not max heaps. We can, never-the-less, convert these two trees into max heaps by percolating the root elements down, as is shown in Figure 5.

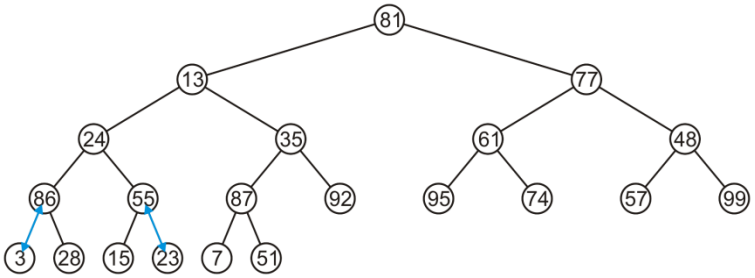


Figure 5. Creating two max-heaps rooted 86 and 55.

Proceeding one row higher, we can swap 48 and 99, 61 and 95, 35 can be swapped with 87 and again with 51, and finally 24 can be swapped with 86 and then 28. Now all the trees rooted at depth 2 are max-heaps, as is shown in Figure 6.

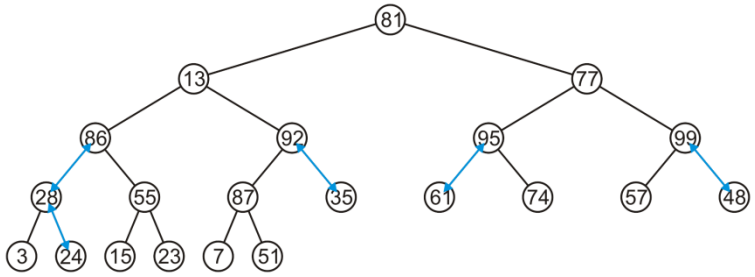


Figure 6. The trees rooted at 86, 92, 95, and 99 are max-heaps.

Continuing back, 77 must be swapped with 99 and 13 must be percolated down to being a leaf node swapping with 92, 87, and 51. This is shown in Figure 7.

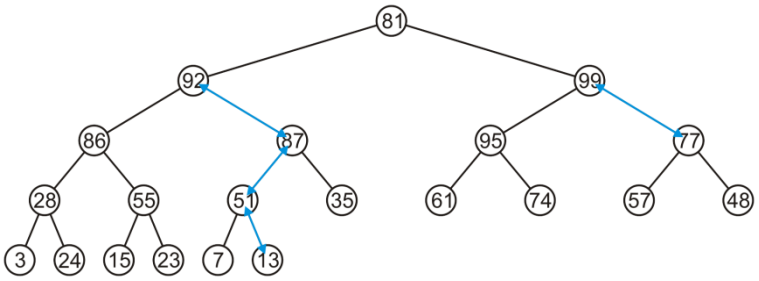


Figure 7. The trees rooted at 99 and 92 are now max-heaps.

Finally, 81 is swapped with 99 and then again with 95. This produces the max-heap shown in Figure 8.

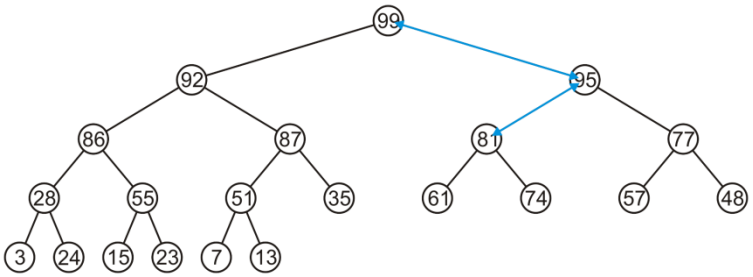


Figure 8. The resultant max-heap.

8.4.3.1 Run-time Analysis

If we consider a perfect tree of height h, there are 2^k nodes at a depth k and a node at depth k will be compared and possibly swapped with at most $h - k$ nodes. Thus, the maximum number of comparisons will be

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

Now, $n = 2^{h+1} - 1$ and $h = \lg(n + 1) - 1$ thus, the run time is $n - \lg(n) = O(n)$.

Note that if we chose to create the max-heap using the same strategy as insertion sort, the run time would be calculated by

$$\begin{aligned}\sum_{k=0}^{h}\left(2^k k\right) &= 2^{h+1}\left(h-1\right)+2\\ &=\left(2^{h+1}+1\right)\left(h-1\right)-\left(h-1\right)+2\\ &=n\left(\lg\left(n+1\right)-2\right)-\lg\left(n+1\right)+4=\Theta\left(n\ln\left(n\right)\right)\end{aligned}$$

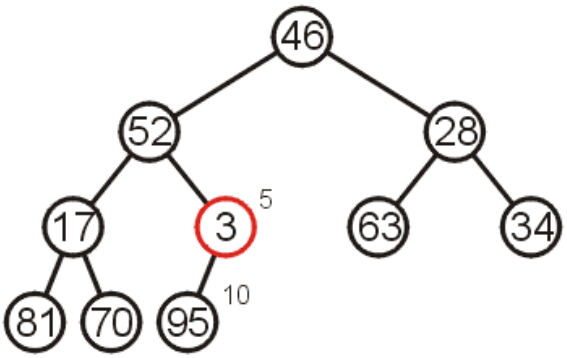
which is significantly worse. Note that this does not affect the overall run time, as the subsequent operation will be $\Theta(n \ln(n))$.

8.4.4 Example

As an example of a heap sort, consider the following unsorted list:

46	52	28	17	3	63	34	81	70	95

First, we must transform the array into a max-heap.



We begin with the entry containing 3 and it has one child so we swap 3 and 95:

46	52	28	17	**95**	63	34	81	70	**3**

Next, the children of 17 are 81 and 70, so we swap 17 and 81:

46	52	28	**81**	95	63	34	**17**	**70**	3

The children of 28 are 63 and 34, so was swap 28 and 63:

46	52	**63**	81	95	**28**	**34**	17	70	3

The children of 52 are 81 and 95, so we swap 52 and 95. In the new location, 52 has one child, 3, and thus no swap is necessary:

46	**95**	63	**81**	**52**	28	34	17	70	**3**

Finally, the children of 46 are 95 and 28, so we swap 46 and 95. In its new position, the children of 46 are now 81 and 52, so we continue by swapping 46 and 81. In that position, the children of 46 are 17 and 70, so we swap 46 and 70.

95	**46**	**63**	81	52	28	34	17	70	3
95	**81**	63	**46**	**52**	28	34	17	70	3
95	81	63	**70**	52	28	34	**17**	**46**	3

Second, we must convert the max-heap into a sorted list:

95	81	63	70	52	28	34	17	46	3

Swap 95 and 3 and percolate 3 into a max-heap of size 9:

81	**70**	63	**46**	52	28	34	17	**3**	95

Swap 81 and 3 and percolate 3 into a max-heap of size 8:

70	**52**	63	46	**3**	28	34	17	81	95

Swap 70 and 17 and percolate 17 into a max-heap of size 7:

63	52	**34**	46	3	28	**17**	70	81	95

Swap 63 and 17 and percolate 17 into a max-heap of size 6:

52	**46**	34	**17**	3	28	63	70	81	95

Swap 52 and 28 and percolate 28 into a max-heap of size 5:

46	**28**	34	**17**	3	52	63	70	81	95

Swap 46 and 3 and percolate 3 into a max-heap of size 4:

34	28	**3**	17	46	52	63	70	81	95

Swap 34 and 17 and percolate 17 into a max-heap of size 3:

28	**17**	3	34	46	52	63	70	81	95

Swap 28 and 3 and percolate 3 into a max-heap of size 2:

17	**3**	28	34	46	52	63	70	81	95

Finally, swap 17 and 3 to produce a sorted list.

3	17	28	34	46	52	63	70	81	95

8.4.5 Black-board Example

Apply heap sort to the list

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

8.4.6 Concluding Remarks

There is no no worst-case scenario—almost lists will be sorted in $\Theta(n \ln(n))$ time. The only case where heap sort will run in $o(n \ln(n))$ time is if all or almost of the entries are equal. The algorithm runs in-place meaning that it requires at most $\Theta(1)$ additional memory. We will contrast this with the algorithms of merge sort and quick sort. Both of these sorting algorithms run $\Theta(n \ln(n))$ time, but both are also, on average, slightly faster. Merge sort, however, requires $\Theta(n)$ additional memory and quick sort requires $\Theta(\ln(n))$, on average.