Heap sort:

Let H = min-ordered binary heap, initially empty
A[1…n] = input array

```
for (k=1; k<=n; k++)
      H.insert (A[k]);
for (k=1; k<=n; k++)
      A[k] = H.removeMin( );
```

Analysis of heap sort:

- Each operation insert, removeMin takes θ(lg n) time.
- So θ(n lg n) total time for n inserts, n removeMins.

In-place Heap sort:

Use a max-ordered binary heap.
Store this heap in the same array A[1..n].
Values in A[1…k] are currently in the heap, so heap size is k.

```
for (k=1; k<=n; k++)
      A.insert (A[k]);
for (k=n; k>=1; k--)
      A[k] = A.removeMax( );
```

1	2	3	4	5	6	7	8
26	48	17	31	50	9	21	16
26							
48	26						
48	26	17					
48	31	17	26				
50	48	17	26	31			
50	48	17	26	31	9		
50	48	21	26	31	9	17	
50	48	21	26	31	9	17	16
48	31	21	26	16	9	17	50
31	26	21	17	16	9	48	
26	17	21	9	16	31		
21	17	16	9	26			
17	9	16	21				
16	9	17					
9	16						
9							

Heap elements highlighted in yellow

Analysis of in-place heap sort:

- Each operation insert, removeMax takes $\theta(\lg n)$ time.
- So $\theta(n \lg n)$ total time for n inserts, n removeMaxes.

Note: there is a faster way to build the heap in only $\theta(n)$ time, but it would still take $\theta(n \lg n)$ time to do the n removeMaxes.

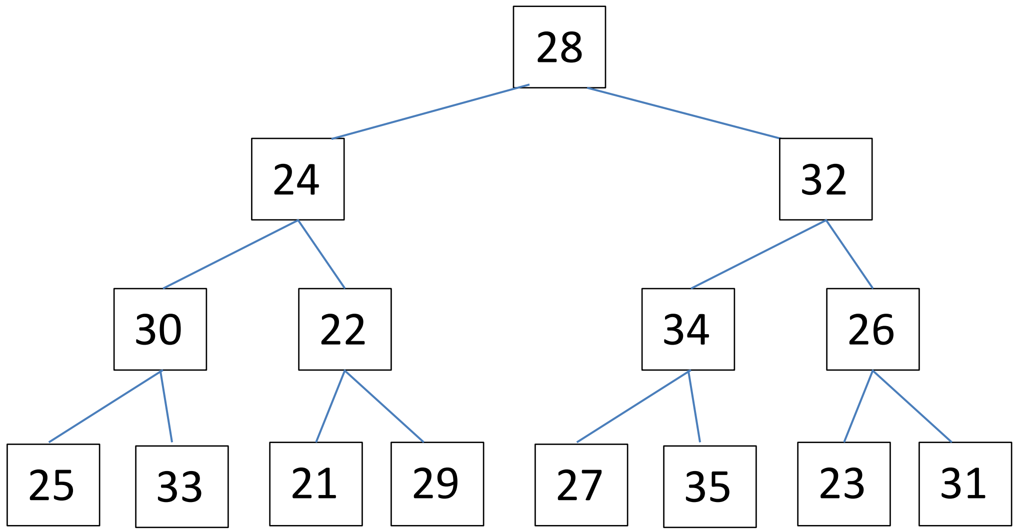
How to build the heap in $\theta(n)$ time?

To build a max-ordered heap in array A[1..n]:

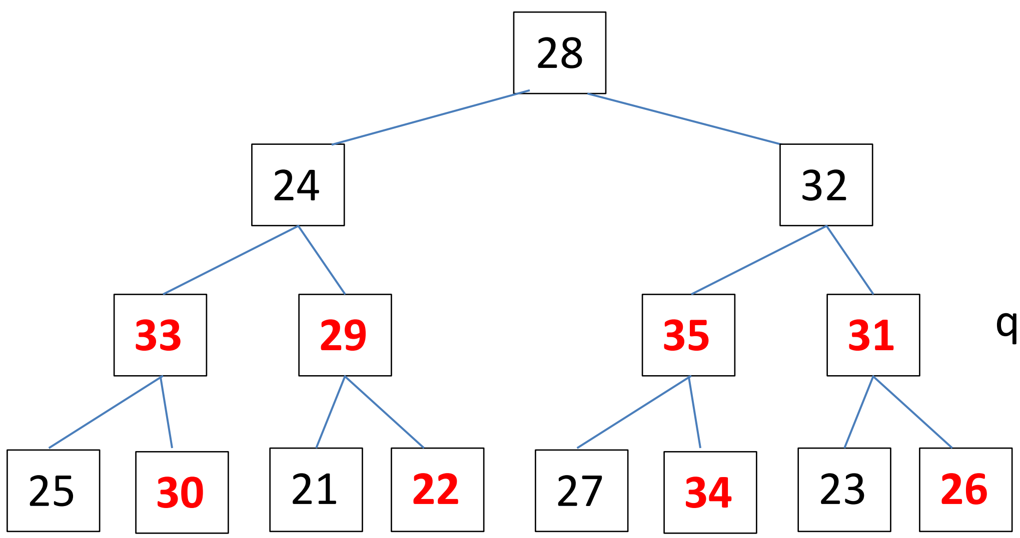
```
Build-Heap (A[1..n]) {
	for (q=n/2; q>=1; q++)
		A.moveDown (q);
}

moveDown (q) {
	while ((2*q <= n  &&  A[2*q] > A[q])
			|| (2*q+1 <= n  &&  A[2*q+1] > A[q])) {
		c = 2*q;
		if (2*q+1 <= n  &&  A[2*q+1] > A[2*q])
			c++;
		swap (A[q], A[c]);
		q = c;
	}
	return x;
}
```

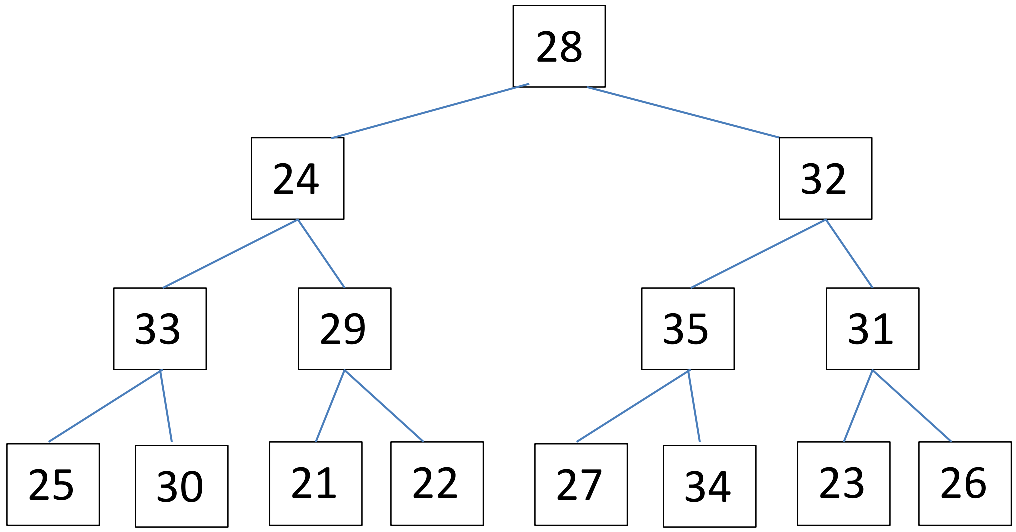
Example:

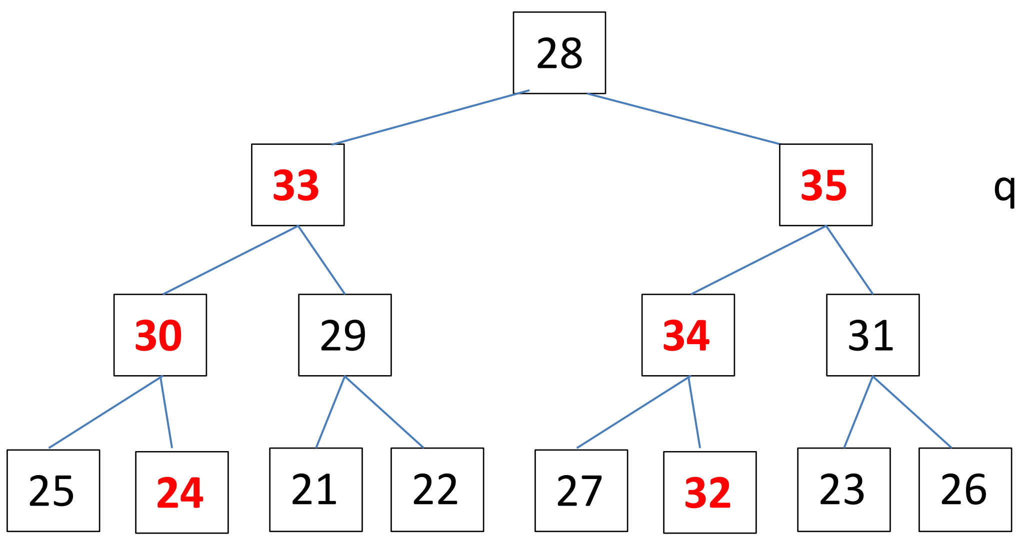


n=15

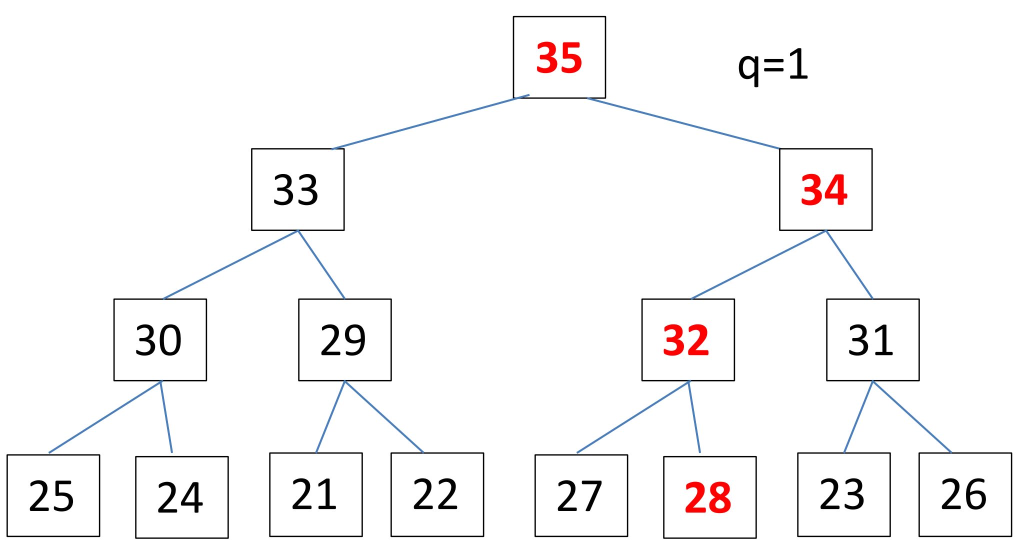


q=7,6,5,4


28
24
32
33
29
35
31
25
30
21
22
27
34
23
26


28
33
35
30
29
34
31
25
24
21
22
27
32
23
26

q=3,2


35
33
34
30
29
32
31
25
24
21
22
27
28
23
26

q=1

Analysis:

In the preceding example with $n=15$,
8 values cannot move down (when $8 \le q \le 15$) (leaf nodes)
4 values move down 1 level (when $4 \le q \le 7$)
2 values move down 2 levels (when $2 \le q \le 3$)
1 value moves down 3 levels (when $q=1$) (root node)

More generally,
$\approx n/2$ values cannot move down (leaf nodes)
$\approx n/4$ values move down ≤ 1 level (parents of leaves)
$\approx n/8$ values move down ≤ 2 levels (grandparents of leaves)
$\approx n/16$ values move down ≤ 3 levels (great-grandparents)
…
1 value moves down $\le \lg n - 1$ levels (root node)

So the total number of swaps is at most:
$1 * n/4 + 2 * n/8 + 3 * n/16 + 4 * n/32 + 5 * n/64 + \ldots$

Reorganize this sum to obtain:

$$
\begin{array}{llllllllll}
 & n/4 & & & & & & & & \\
+ & n/8 & + & n/8 & & & & & & \\
+ & n/16 & + & n/16 & + & n/16 & & & & \\
+ & n/32 & + & n/32 & + & n/32 & + & n/32 & & \\
+ & n/64 & + & n/64 & + & n/64 & + & n/64 & + & n/64 \\
 & \ldots & & \ldots & & \ldots & & \ldots & & \ldots
\end{array}
$$

Each above column is an bounded by an infinite geometric sum with ratio ½.

The sum of an infinite geometric series with first term f and ratio r is f/(1–r).

Summing each column yields:

$$n/2 \quad + \quad n/4 \quad + \quad n/8 \quad + \quad n/16 \quad + \quad n/32 \quad + \quad \ldots$$

But this is another infinite geometric sum with ratio ½.
Its sum is $(n/2) / (1–½) = n$.

Therefore the total number of swaps in Build-Heap is at most n, and the worst-case running time of Build-Heap is $\Theta(n)$.