Given this implementation, how do operations work?

* **insert(x)**: (1) insert x at next free index.
  (2) if heap order maintained, done.

\[ \text{O/w swap } x \text{ with parent until heap order satisfied} \]
  [Percolate up]

**Example:**

```
  13
 / \  \
 21 16  \\
 / \  \\
24 31 19 68
 / \  \
65 26 35 14
```

**insert(35): OK**

**insert(14):**

**Runtime:** in terms of \( n \) = # of nodes

- **worst:** \( O(\log n) \) (height of tree)
- **avg.:** \( O(1) \) Percolation typically ends early.

* **deleteMin():** (recall that min is always at root).
  (1) Save root as temp
  (2) move last element \( y \), at root.
  [Percolate down]
  (3) return temp
DeleteMin()

Final

RunTime:
- worst: $O(\log n)$
- avg: Also $O(\log n)$, unlikely that last element will be min

Other operations
- BuildHeap(): given n elements, create a min heap

Easy way: n consecutive inserts.
- RunTime: worst: $O(n \log n)$
- avg: $O(n)$
Better: (1) insert elements in any order that maintains structure property
(2) starting at level above deepest leaves, up to root
for each level:
  for each node $x$, at this level:
    percolate $x$ down (swap up/smaller)
    until $x$ satisfies heap order

ex: 50, 60, 20, 15, 7, 25, 35, 40, 10, 5, 30

(1): 50
    /
   /  
  60   20
      /
     /  
    15   7
       /
      /  
     40   10
        /
       /  
      5     30

(2): 50
    /
   /  
  60   20
      /
     /  
    15   7
       /
      /  
     10   7
       /
      /  
     25   35

Final: 5
      /
     7   20
        /
       /  
      10   30
         /
        /  
     25   35
        /
       /  
     40   15
       /
      60   50
         √
Recursive function

\[ \text{BuildHeap(e)} \] // e is initially root

if e not leaf

\[ \text{BuildHeap(left child of e)} \]
\[ \text{BuildHeap(right)} \]
\[ \text{PercolateDown(e)} \]

Run Time:

Seems like $O(n \log n)$

Percolate $n$ nodes, each percolation takes $\log n$.
* But actually time is $O(n)$!

Beautiful proof:

1. Start here ->
2. Next here ->
3. Last here ->

Run time $\sim$ # swaps.

Suppose always swap with left-most unswapped descendant.

Cross out node when we swap with it:

# x's = # swaps $\sim$ runtime.

How many x's? $n - \log(n) = O(n)$. 
Notice: can’t start at top and swap w/ smaller

\[ \begin{array}{c}
\text{10} \\
\text{12} \quad \text{9} \\
\text{14} \quad \text{6} \quad \text{5} \quad \text{8}
\end{array} \ \Rightarrow \ \begin{array}{c}
\text{9} \\
\text{12} \quad \text{10} \\
\text{14} \quad \text{6} \quad \text{5} \quad \text{8}
\end{array} \ \Rightarrow \ \begin{array}{c}
\text{9} \\
\text{12} \\
\text{14} \quad \text{6} \quad \text{10} \quad \text{8}
\end{array} \]

\[ \uparrow \]

Can already see \(9 \leq 6\)

violate heap order property \((9 > 6)\)

Another operation

Decrease Key \((x, \text{value})\) - decrease \(x\)'s priority to value

ex: Decrease Key \((25, 18)\) (from BuildHeap() example)

How to implement in \(O(\log n)\)? Need to find \(x\).

\[ \rightarrow \text{Keep look-up (hash) table of elements + their indices.} \]

\[
\begin{array}{c|c}
\text{x} & \text{i} \\
25 & 6 \\
40 & 8 \\
\vdots & \vdots \\
\end{array}
\]

- find index, \(i\), of \(x\) in table.
- set \(\text{array}[i] = \text{value}\)
- percolate \(\text{array}[i]\) up until heap order satisfied
ex. \( i = 6 \)

\[
\begin{array}{c}
5 \\
7 \quad 20 \\
10 \quad 30 \quad \boxed{18} \quad 35 \\
\end{array}
\Rightarrow
\begin{array}{c}
5 \\
7 \quad 18 \\
\ldots \quad 20 \quad 35 \\
\end{array}
\]