Priority Queues

Suppose various maintenance jobs need to be tended to:
- Fix leaking roof. Priority: 3
- Printer. Priority: 5
- Release skunk. Priority: 1

In what order should we process these?
Use a queue? First-in-first-out? No (based on priority)

Priority Queue - queue where elements have a priority by which they are removed.

(Some) Operations:
- `insert()`: (insert a new element into the queue)
- `deleteMin()`: remove & return the element with lowest priority value (i.e., highest priority element).

Implementations:
- `insert`: O(1), `deleteMin`: O(n)
- Sorted LinkedList: O(n), O(1)
- AVL Tree: O(log n), O(log n)

... has other operations (contains(), traversals) that we don't need. So, we use a new data structure.

- Minimum Binary Heap (Min-Heap) (Max-Heap also exists)
- Visualize as a tree
- Has 2 requirements
(1) **Heap order property** - each node has priority ≤ both of its children.

```
      20
     / \```
```
    10   30
   /  \
  5  15  25 40```

**AVL**

```
      5
     / \
    10  15
   /  \
  20  30  25 40```

**Min-Heap**

(2) **Structure property** - levels are filled from left to right, every level (except possibly last) is completely filled.

```
AVL:
```
```
  new insert```

Heap:

```
  new insert```

How to implement?

*Simplest way to store is using 1D fixed size array.*

**Array Implementation (1D, sufficiently large)**

```
0 1 2 3 4 5 6 7
```
```
| 5 | 10 | 15 | 20 | 30 | 25 | 40 |
```

<See why this is blank soon>

But should be able to easily access children of each node. Can we?

Yes! For element at index `i`: left child in `2i`

```
right = 2i + 1
parent = i/2```

Given this implementation, how do operations work?

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

O(log n) swap x with parent until heap order satisfied
Percolate up.

Ex: 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.

Swap y with larger child until heap order satisfied
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: $O(n)$