Priority Queues

Suppose various maintenance jobs need to be tended to:
- Fix leaking roof \( \text{Priority} = 3 \)
- Printer \( \text{Priority} = 7 \)
- Release Skunk \( \text{Priority} = 1 \)

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 and return the element with lowest priority value (i.e., highest priority element).

Implementations:

<table>
<thead>
<tr>
<th>Data Structure</th>
<th><code>insert</code></th>
<th><code>deleteMin</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkedList</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Sorted LinkedList</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>AVL Tree</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
</tbody>
</table>

AVL Tree 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 \( \leq \) both of its children.

\[
\begin{array}{c}
\text{AVL} \\
20 \\
10 \quad 30 \\
5 \quad 15 \quad 25 \quad 40 \\
\end{array}
\begin{array}{c}
\text{Min-Heap} \\
\ 5 \\
10 \quad 15 \\
20 \quad 30 \quad 25 \quad 40 \\
\end{array}
\]

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

\[
\begin{array}{c}
\text{AVL} \\
\text{new insert} \\
\end{array}
\begin{array}{c}
\text{Heap} \\
\text{new insert} \\
\end{array}
\]

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
\[
\begin{array}{cccccccc}
5 & 10 & 15 & 20 & 30 & 25 & 40 & \_ \\
\end{array}
\]
```

(See why yours is blank soon)

Should be able to easily access children/parent of each node.

How? For element at index \( i \):

- Left child: \( 2i + 1 \)
- Parent: \( \lfloor i/2 \rfloor \)
Given this implementation, how do operations work?

- **insert** $(x)$: (1) insert $x$ at next free index. (2) if heap order maintained, done. o/w, swap $x$ with parent until heap order satisfied. Percolate up.

Ex: 13

```
21   16
24   31   19   68
```

Final

```
13
21   16
24   31   19   68
```

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.
  3. swap $y$ with larger child until heap order satisfied
  4. Percolate down
  5. 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

\[\begin{array}{c}
\text{(1): } 50 \\
\quad \leftarrow \\
\quad 60 \quad 20 \\
\quad \quad \leftarrow \\
\quad 15 \quad 7 \quad 25 \quad 35 \quad \Rightarrow \\
\quad 40 \quad 10 \quad 5 \quad 30 \\
\end{array}\]

\[\begin{array}{c}
\text{(2): } 50 \\
\quad \leftarrow \\
\quad 60 \quad 20 \\
\quad \quad \leftarrow \\
\quad 15 \quad 7 \quad 25 \quad 35 \quad \Rightarrow \\
\quad 40 \quad 10 \quad 5 \quad 30 \\
\end{array}\]

\[\begin{array}{c}
\text{Final: } 5 \\
\quad \leftarrow \\
\quad 20 \\
\quad \quad \leftarrow \\
\quad 10 \quad 7 \quad 25 \quad 35 \quad \Rightarrow \\
\quad 40 \quad 15 \quad 60 \quad 30 \\
\end{array}\]
Recursive function.

\text{BuildHeap}\ (e) \quad //\ e\ is\ initially\ root
\begin{align*}
\text{if } e \text{ not leaf} & \rightarrow \\
\text{BuildHeap}\ (\text{left child of } e) & \\
\text{BuildHeap}\ (\text{right child of } e) & \\
\text{Percolate-Down}(e) & \\
\end{align*}

\textbf{Runtime:}

\text{Seems like } \Theta(n \log n)

Percolate \( n \) nodes, each percolation takes \( \log n \)

\* But actually time is \( O(n) \)!

\textbf{Beautiful proof:}

1. Start here \( \rightarrow \)
2. Next here \( \rightarrow \)
3. \text{Run time } \sim \# \text{swaps}
4. Suppose always swap with left-most unswapped descendant
5. Cross out node when we swap with it
6. \# \text{x's} = \# \text{swaps} \sim \text{runtime}

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

Another operation
Decrease Key (x, 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.

⇒ Keep look-up (hash) table of elements + their indices (key, element)

(key) (index)

40 8 - percolate array[i] up until heap order satisfied

25 6 - update table (at most O(log n) updates)