Binary Heaps

- Binary heaps are used for two purposes:
  - Priority Queues
  - Heap sort
Binary Heaps

- Priority Queue
  - A priority queue is where you always extract the item with the highest priority next.

- Priority Queue Example
  - Let’s say we are Google and we want an efficient way to do determine which applicant from our applicant pool to interview when a new position opens up.
  - So we assign a priority based on a particular formula – including application arrival time, GPA, and understanding of Heaps, ironically enough.
How could we implement this using our existing methods?

- We don’t want just a normal queue, because that’s FIFO, doesn’t care about a priority value.
- We could use a linked list sorted by priority.
  - Then we would have a long insertion time for insert, because we have to traverse the list to find where our element goes.

This isn’t necessary, because all we care about is the next applicant to interview, not that the list is sorted.
Consider a minimum binary heap:

- Looks similar to a binary search tree
- BUT all the values stored in the subtree rooted at a node are greater than or equal to the value stored at the node.
The only operations we need are:

- **Insert** and **RemoveMin**
- We can implement a heap using a complete binary tree or an array as we will talk about later.
- No matter how we implement it, we will visualize the data structure as a tree, like the one above.
Insert

- Since we want a complete binary tree
  - We insert the new node into the next empty spot
  - Filling each level from left to right
  - Then we need to worry about where this node should move to depending on its priority.
The problem is in all likelihood, if the insertion is done in this location, the heap property will not be maintained.

Thus, you must do the following "Percolate Up" procedure:

- If the parent of the newly inserted node is greater than the inserted value, swap the two of them.
- This is a single "Percolate Up" step.
- Now, continue this process until the inserted node 's parent stores a number lower than it.
**Insert**

- **Percolate Up:**
  - If the parent of the newly inserted node is greater than the inserted value, swap the two of them.
  - Now, continue this process until the inserted node 's parent stores a number lower than it.

```
<table>
<thead>
<tr>
<th></th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Steph</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Sarah</td>
</tr>
<tr>
<td>35</td>
<td>200</td>
</tr>
<tr>
<td>Sally</td>
<td>Otto</td>
</tr>
<tr>
<td>100</td>
<td>Ken</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Mags</td>
</tr>
<tr>
<td>45</td>
<td>Al</td>
</tr>
</tbody>
</table>
```
Heap Implementation

- **Array Implementation:**
  - Instead of using a binary tree implementation,
  - We can use an array implementation where the children of the node at index $i$ are the nodes at indices $2i$ and $2i+1$.

### Example

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>...n</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>X</td>
<td>2</td>
<td>35</td>
<td>5</td>
<td>200</td>
<td>100</td>
<td>10</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Steph</td>
<td>Sally</td>
<td>Sarah</td>
<td>Otto</td>
<td>Ken</td>
<td>Mags</td>
<td>Al</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of a heap with nodes and indices]
Delete Minimum

- Delete the min (which is always the root), and return:
  - Now we need to replace it, but with what?
    - Replace with the last element in the array, or the last node added to the tree.
  - Then Percolate Down.

Percolate Down:
If the children of this node has children less than it swap it with the MIN of its 2 children, until the node has children that are larger than it.
Runtime of heap operations

- What is the height of a complete binary tree?
  - (Where we completely fill each level from left to right)
  - The maximum height is \( h = \log_2((n+1)/2) \)
- Insert
  - So for insert the maximum number of times we can swap a node up the tree in a tree with \( n \) nodes would be
    \[ \log_2((n+1)/2) \]
- DeleteMin
  - And for delete the maximum number of times we can swap a node down the tree in a tree with \( n \) nodes would be
    \[ \log_2((n+1)/2) \]
- So, Insert and Delete are both \( O(\log n) \)
Heapify

- Bottom up heap construction
  - How to construct a heap out of an unsorted array of elements.
    - 1) Place all the unsorted elements in a complete binary tree.
    - 2) Go through the nodes of the tree in backwards order running Percolate Down on each of these nodes. (Skip over all leaf nodes.)
- Shown on the board
Heapify

- Bottom up heap construction
  - As this is done, one invariant we see is that each subtree below any node for which Percolate Down has already executed is a heap.
    - Thus, when we run Percolate Down on the root at the end of this algorithm, the whole tree is one heap.
  - Can you see why we can NOT go through the nodes in forward order?
    - Give an example where doing so produces a tree that is not a heap.
Heapify Analysis

- Shown on the board
Heapsort

- Shown on the board
  - Now that we have determined how to execute several operations on a heap, we can use these to sort values using a heap. Here is the idea:
    - 1) Insert all items into a heap
    - 2) Extract the minimum item n times in a row, storing the values sequentially in an array.
  - Since each inserting and extraction take $O(lg n)$ time, this sort works in $O(n lg n)$ time.