The Heap data structure

Often used for

- Implementing “priority queues” and “heapsort” (an $O(n \log n)$ sort)

Based on a binary tree ...

- each “node” contains a value, plus a left and right child

Min Heap main idea ...

- each node has a smaller value than its children

  ... heap constraint

- Q: what value does the root have?

  ... the smallest value

• a “complete” binary tree (filled left-to-right)
  - insert:
    1. add node to next open location
    2. “trickle up” as needed to maintain heap constraint

- delete:
  1. replace root value with last heap value
2. delete last heap node
3. “trickle down” as needed to maintain heap constraint

You’ll need to define helper functions ...

For `deleteMin`
- `lastChild` ... to return the value of the last child
- `deleteLast` ... to delete the last node in the Heap
- `trickleDown`

For `insert` (probably the hardest function)
- `isFull` ... to check if tree is full (for finding empty spot)
- `height` ... to get the height of tree (for finding empty spot)
- `trickleUp`

Basic idea for `insert` before `trickle up`
- assuming the left and right subtrees aren not empty —
- insert into the left subtree if:
– left subtree isn’t full
– left and right are same height and right subtree is full

• otherwise insert into right subtree

Note there are other ways to do insert as well ...

One way to do isFull

\[
\text{isFull} \ colon \ \text{Tree} \ a \ \rightarrow \ \text{Bool} \\
\text{isFull} \ \text{Nil} \quad = \ True \\
\text{isFull} \ (\text{Node} \ \_ \ \text{Nil} \ \text{Nil}) \ = \ True \\
\text{isFull} \ (\text{Node} \ \_ \ \text{l} \ \text{Nil}) \ = \ False \\
\text{isFull} \ (\text{Node} \ \_ \ \text{Nil} \ \text{r}) \ = \ False \\
\text{isFull} \ (\text{Node} \ \_ \ \text{l} \ \text{r}) \quad = \ \text{isFull} \ \text{l} \ \&\& \ \text{isFull} \ \text{r} \ \&\& \ \text{height} \ \text{l} \ == \ \text{height} \ \text{r}
\]