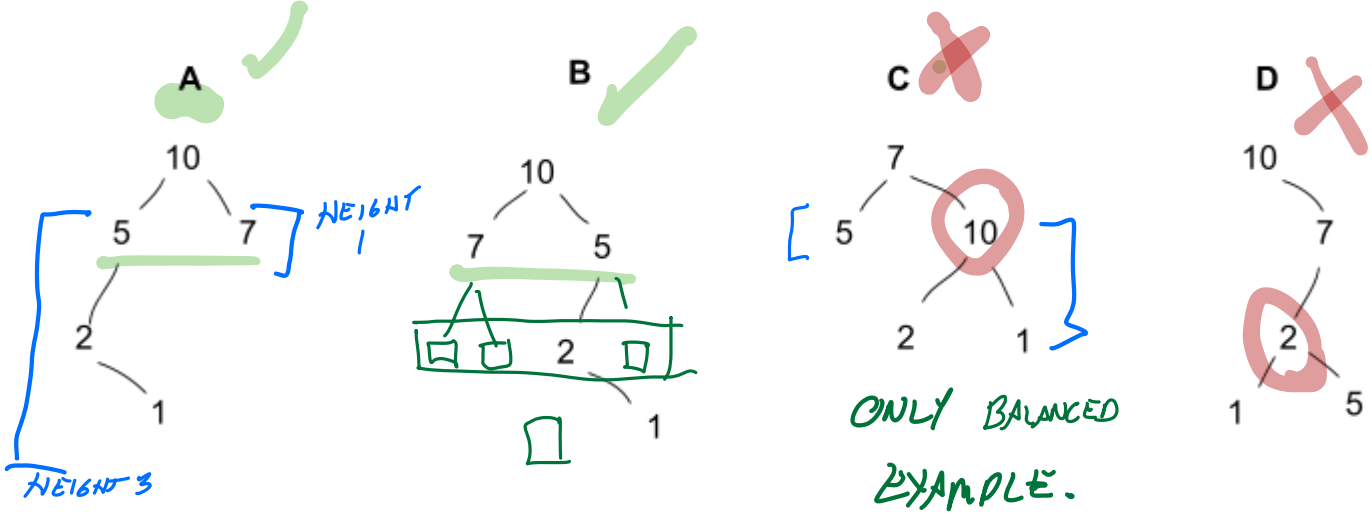
Heap: A binary tree in which the highest-priority item is at the root and both the left and right subtrees are also heaps

```
Key operations:
 - get_max:  constant time (just look at max)
 - insert:  Add a new item => O(logN) for N items,
if the tree is balanced
 - remove_max:  O(logN) if tree is balanced
```

Exercise: Which of the following are heaps? Which are balanced (whether or not they are heaps)?



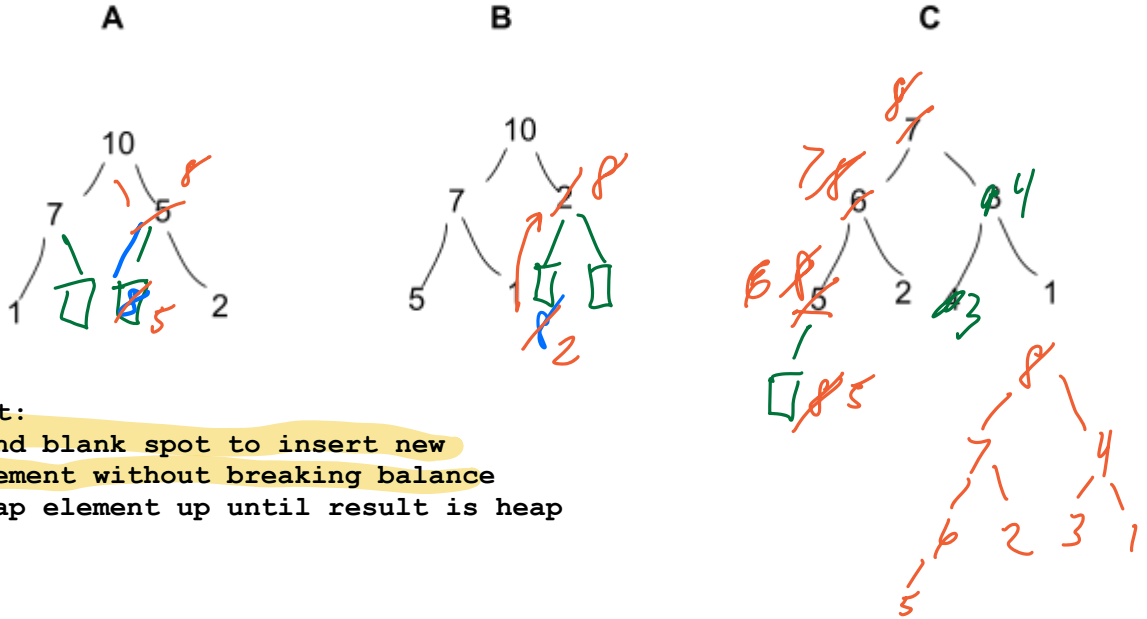
Define "balance":

```
at any node, height of left and right subtrees
differ by at most 1
Or:  "How I filled up every row of the tree other
than the bottom-most one?"
```

Goal: implement heaps with the run-times stated above.

Try it: insert 8 into each of the following trees, while maintaining requirements

```
Requirements:
 - Result must be a heap
 - Can only modify one
branch of the tree
```




```
insert:
1. Find blank spot to insert new
   element without breaking balance
2. Swap element up until result is heap
```

Implementation: here's a binary-tree class in Python

Aside on Python syntax: these are "keyword arguments" => arguments to a function specified by name. If you leave them out, default value is used (here, None)
=> Can have functions with optional arguments!

```
class BinTree:
    def __init__(self, data, left=None, right=None):
        self.left = left
        self.right = right
        self.data = data
```

```
# a         unbalanced_tree = BinTree("a",
#  \                                  left=None,
#   b                                 right=BinTree("b",
#  /  \                                             left=None,
#    c                                              right=BinTree("c",
#   /                                                            left=BinTree("d"),
#  d                                                             right=None)))
```

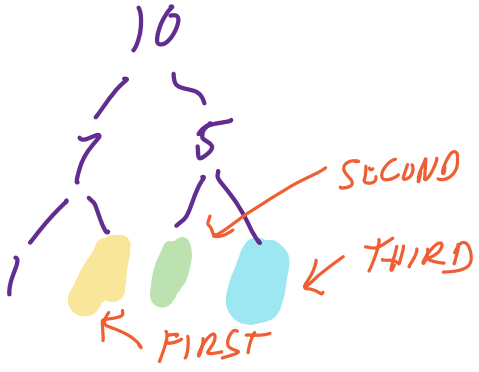
insert:

1. **Find blank spot to insert new element without breaking balance**
   => Need way to know which spots are empty, would need to store some extra info

1. **Swap element up until result is heap**
   => Need to find node "above" you => requires a doubly-linked tree (field for parent) (HW2)

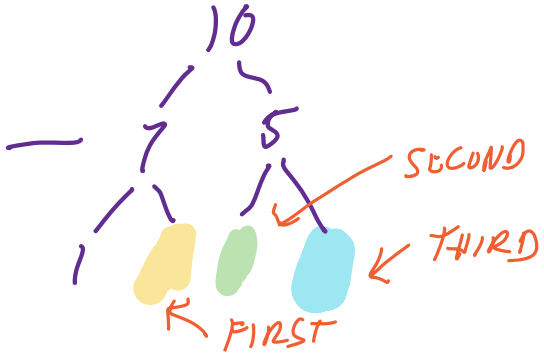
Usually, use a different kind of representation that:
- **Is easy to find an open slot**
- **Is easy to navigate up and down tree**

To simplify: enforce that all inserts use the next available slot in the last row of the tree (don't have to choose)

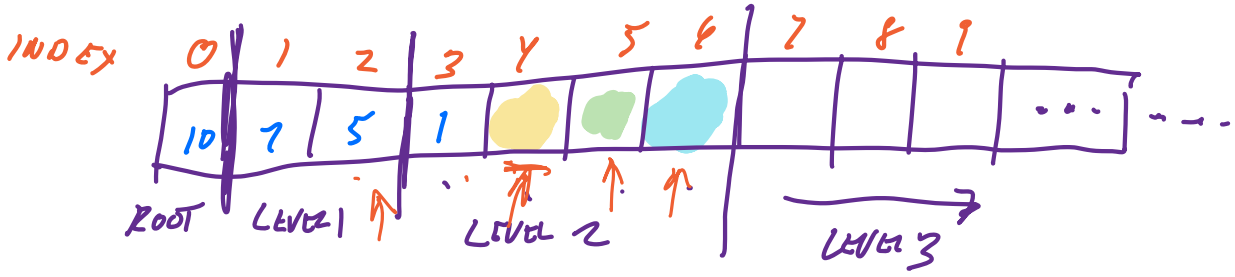


How to create an easier implementation?

To simplify: enforce that all inserts use the next available slot in the last row of the tree (don't have to choose)

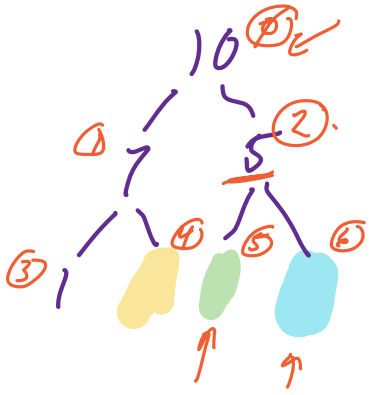


SOLUTION: EMBED TREE INSIDE AN ARRAY



Easy to find next open spot: just look for first element that's empty

ORANGE = ARRAY INDICES



For some node at index i:
- left(i) is at (i * 2) + 1
- right(i) is at (i * 2) + 2
- parent(i) is at floor((i - 1)/2)

Ex. i = 2 HOLDS 5

LEFT = 2*2 + 1 = 5

RIGHT = 2*2 + 2 = 6

PARENT = $\left\lfloor \frac{2-1}{2} \right\rfloor = \left\lfloor \frac{1}{2} \right\rfloor = 0$

We'll pick up from here next lecture. . .