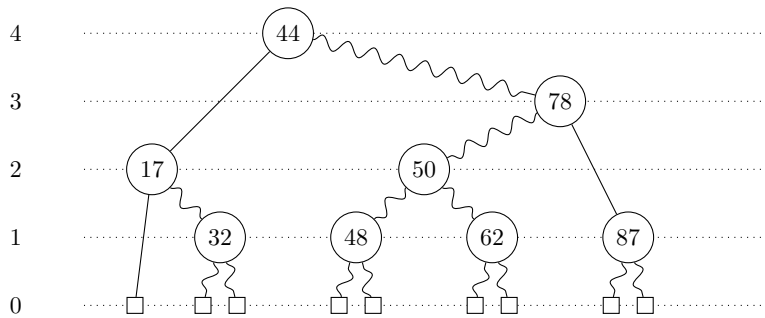
In order to maintain balance in binary search trees, we are going to add the concept of each node having a *level*, which is similar to the height of a tree that you read about. A `nullptr` is at level zero, and *only* `nullptr` are at level zero. Every leaf node is at level one. Every non-`nullptr` node will have a positive level, and that level will be *greater than* the larger of the levels of its children.

We say that the *length* of an edge from parent to child is the difference in the levels of the two nodes. If we say a node in our tree *has shape* (x, y), this means the node's left child is x layers lower and its right child is y layers lower. We do not define a shape for `nullptr` nodes.

Question 1. What is the shape of each node in the following tree? Layers are drawn horizontally.



We will modify the insert and delete procedures of a binary search tree to produce a *Level-Balanced* tree. A tree is *level-balanced* if every edge is of length 1 or 2.

We will maintain this *level-balanced* property as a **class invariant**. The level-balanced property may disappear *during* an insert or delete procedure, but we cannot return from such function calls until we have fixed the problem. We cannot refuse an insert or delete; the set of keys in the tree must be the same as if we were not balancing.

Let's look at an insert sequence that, with a "standard" binary search tree, would produce an imbalanced tree. As we insert each one, we notice where an imbalance occurs, and we will fix it.

Question 2. Into an initially empty level-balanced binary search tree, insert the keys $20, 19, 18$ in that order.

Question 3. Into the resulting tree from the last part, insert 12 then 9, then 17.

Question 4. Insert 7 then 5 into the resulting tree from the previous question.

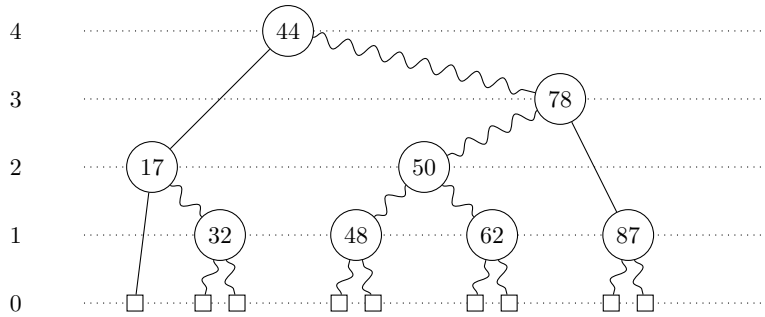
Question 5. Insert 13 then 15 into the resulting tree from the previous question.

Question 6. Insert 6 then 16 into the resulting tree from the previous question.

Question 7. Insert 14 into the resulting tree from the previous question.

Deletion from a Level-Balanced Tree

Let's recall the tree we began the first lesson on this topic discussing.



Question 8. Designate a vertex or vertices we can delete without needing to move the layer of any other vertices.

Question 9. Suppose we delete 87 from the original tree. What happens to the tree?

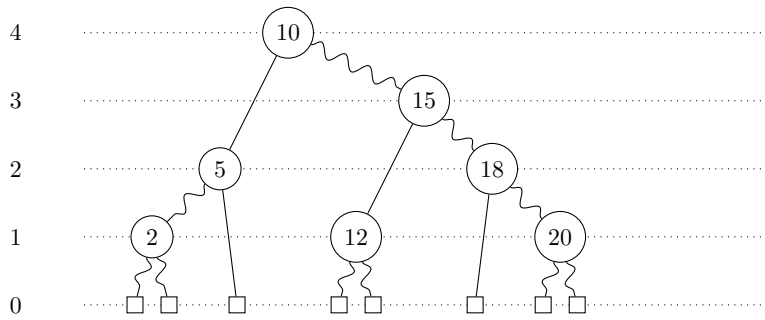
Question 10. What if we delete 62 from the original tree, *then* delete 87?

Question 11. From the original, delete 48 **and** 62. What happens?

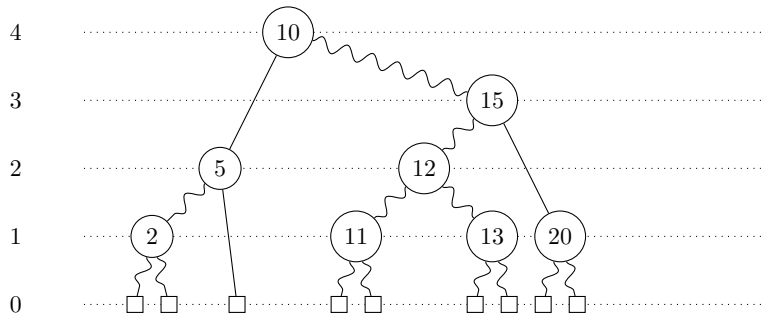
Question 12. Take the result of *those* deletions and delete 87. What happens now?

Question 13. Suppose we delete 50 from that result. What happens now?

Question 14. Suppose we delete key 2 from the following tree. What happens?

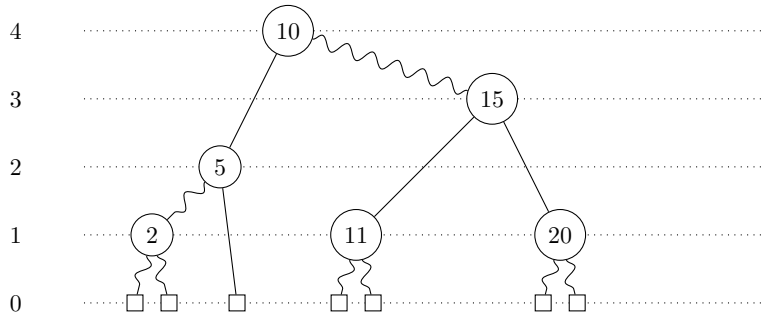


Question 15. Suppose we delete 2 from the following tree. What happens?



Hint: Consider also the *same* case, but if every node except two were one layer higher, and the current leaf nodes had at least one child each. This is also important as it ensures your rule generalizes.

Question 16. Suppose we delete 2 from the following tree. What happens?



The Height of a Balanced Tree

Recall that a tree's **height** is the maximum edges from the root to any leaf. A leaf has height zero. We will show that by height that this type of tree achieves height balancing.

Question 17. Prove the following claim: if we balance a tree using these rules, a tree with n keys has height $\mathcal{O}(\log n)$

Question 18. We are going to create a balanced binary search tree with these rules. What is the fewest nodes you can have to achieve a height of one? Two? An arbitrary value of n?