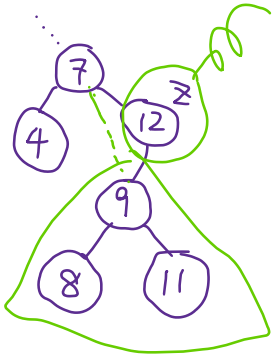
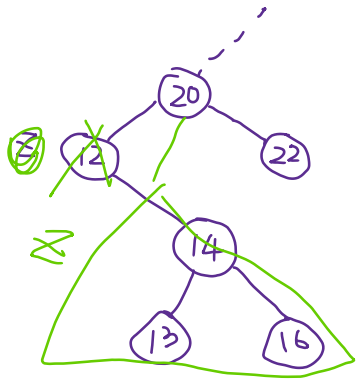
More Binary Search Tree Methods

- Deletion
  - Here, we assume that we are about to delete an **existing unique** item from a binary search tree. Our algorithm will search in the binary search tree for the node containing the item first, so if the item is not unique, the algorithm deletes the node that contains the first found item.

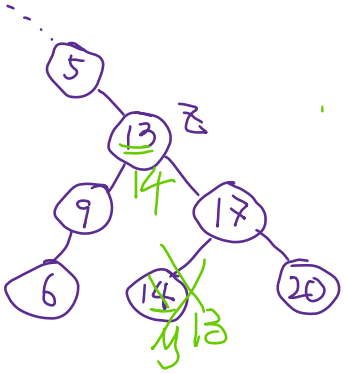
```
tree_deletion (x, item)        # We are deleting item from the tree rooted at node x
1  z = tree_search(x, item)    # item exists in T so z cannot be a None node
2  if z is a leaf:
3        z = None
4  else if z has exactly one child:
5        z = z's child
6  else:
7        y = node contains tree_minimum (z.right)
8        swap z.val and y.val  # z.val = item before swapping
9        tree_deletion (y, item)
```

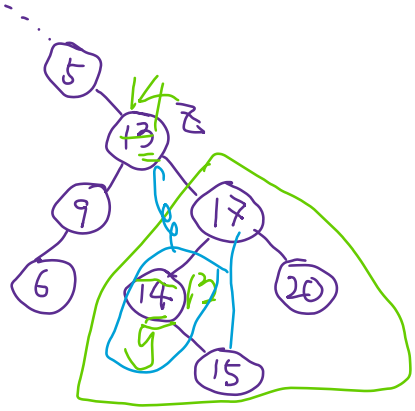
  - When $z$ has no children, we can simply delete it without violating the binary-search property.
  - When $z$ has only one child:




  - When $z$ has two children, the node with minimum item in $z$'s right subtree must not have a left child (why?). Thus, when we call **tree_deletion** $(y, item)$, $y$ must have either $0$ or $1$ child. $y$ contains the next larger item after $item = z.val$ in the sorted order.



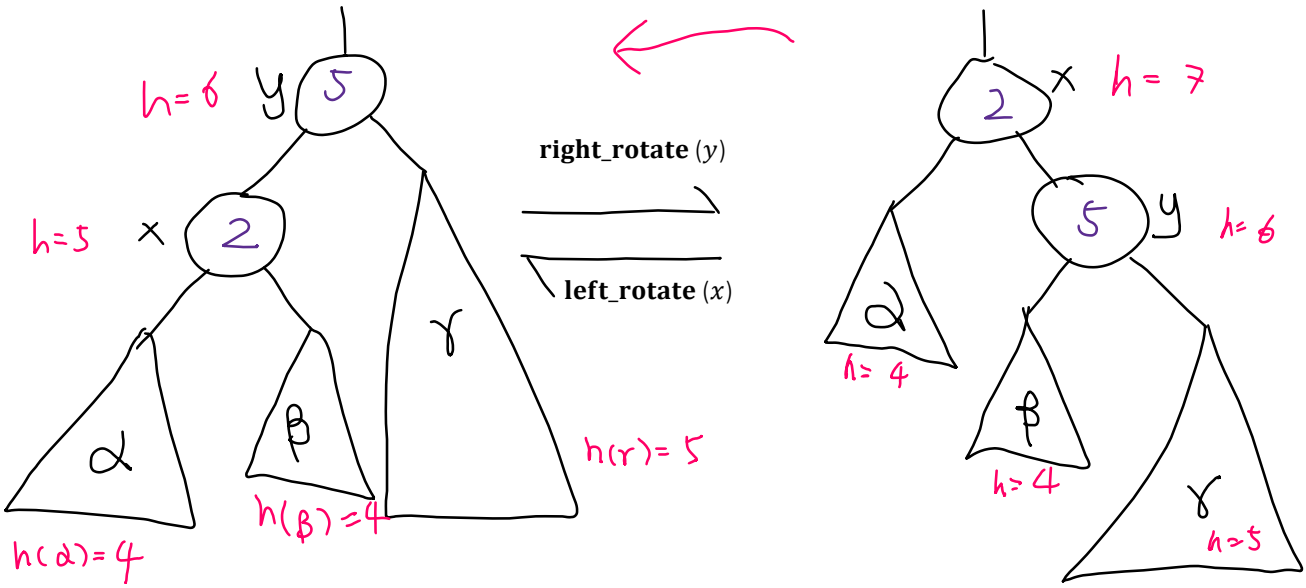



- Similar to search and insertion, **tree_deletion** also takes $O(h)$ time, since we go down from the root node following one path.

1. In the line 7 of the **tree_deletion** algorithm, what if we let $y =$ node contains **tree_maximum** $(z.left)$? That is also okay, since now $y$ contains the immediate previous item comparing to $item = z.val$ in sorted order.

Reduce tree height

- When we have a binary search tree with large height, how to reduce its height while keeping the binary-search property? We use a method called rotation.




- Here, $\alpha, \beta$ and $\gamma$ are subtrees, they can be empty, they can be just one node and they can be a large tree.
- On the left we have $\alpha < 2 < \beta < 5 < \gamma$, and on the right we also have $\alpha < 2 < \beta < 5 < \gamma$; thus binary-search property is kept.
- The running time of rotation is $\Theta(1)$, since we only need to reconnect a constant number of links.
- Let $h(\alpha), h(\beta), h(\gamma)$ be the height of subtree $\alpha, \beta$ and $\gamma$ respectively, if $h(\gamma) > h(\alpha)$ and $h(\gamma) > h(\beta)$, then left rotation can reduce the height of this subtree by $1$.
- We don't **left_rotate** a node whose right child is **None**, and we don't **right_rotate** a node whose left child is **None**.

Balanced Binary Search Trees

- Remind that, a binary tree is balanced if its height is $\Theta(\lg n)$. Since search, insertion and deletion all have time complexity $O(h)$, if we can bound the height of a binary search tree to $\Theta(\lg n)$ then all these methods have running time $O(\lg n)$.
- There are many designs of balanced binary search trees, such as AVL tree, red-black tree, splay tree… all these trees use rotation to fix its height. In our class, we introduction AVL trees.