Q2.(ii)

In general, this idea should not be recommemded since the complexity is much larger than the original idea.
For the original *delete* operation, whatever the condition is, the complexity is always $O(\log_2 n)$. First, we need $O(h)$ steps to find the node to delete at height h. Then, we have two conditions:

- If the right successor has a non-empty left successor tree, then we need to merge this tree with left successor tree. For the worst case, we need to search k times, where k is the height of this non-empty left successor tree. So the complexity is in $O(k)$. The overall complexity is $O(h + k) = O(\log_2 n)$.
- For other cases, we need constant time to transplant new subtree into the place where the deleted node occupies, the overall complexity is just $O(h)$.

So the complexity of *delete* operation is $O(log_2 n)$.

But for the *_delete_with_reinsert* operation, the situation becomes different. First, we still need $O(h)$ steps to find the node to delete at height h. Then, if the number of nodes in the left and right subtrees is m, we need to reinsert them one by one. Luckily all the nodes in the subtrees are bigger or smaller than the parent of the deleted node, so we can just insert starting from that parent node.
For the worst case, we reinsert all the nodes linearly, then the complexity is in $O(m^2)$. Even for average case when the tree after reinsertion is almost balanced, the time complexity is still in $O(m)$ for building a binary tree. So the overall complexity is either $O(h + m^2)$ or $O(h + m)$, which is greater than $O(h + k) = O(\log_2 n)$.
Besides, we have not taken the *free* operations for the nodes when reinserting and the space complexity for creating new nodes into consideration.
So considering the time complexity and space complexity, *_delete_with_reinsert* operation is not recommemded.