In the fourth lecture about binary tree, we will talk about how to delete a node in a binary search tree, and still keep the new tree as a binary search tree.

Consider this binary search tree. We want to delete only one node and the result is still a binary search tree.

This problem can be divided into three cases: In the first case, the node to be deleted has no child. This is the easiest case. In the second case, the node to be deleted has only one child.

In the third case, the node to be deleted has two children.

We will handle these three cases separately.

Consider the easiest case: the node to be deleted has no child. In other words, the node to be deleted is a leaf node. This binary search tree has many leaf nodes, including 6, 15, 29, 36, 46.

To delete one of these leaf nodes, we need to set the parent’s pointer to NULL and then free the memory.

In the second case, the node to be deleted has one child. In this binary search tree, 3 is a node that has only one child.

If we want to delete node 3, we will make 3’s parent points to 3’s only child. 3’s parent is node 9. 3 is node 9’s left child. 3’s child is node 6. We will make node 9’s left child point to node 6. Then, we need to free the memory of node 3.

If the node to be deleted has two children, it is more complex. Consider the situation when we want to delete node 27 and still keep this as a binary search tree.

We are going to find the immediate successor of 27, swap them, and then delete 27 from the right subtree.

Let’s go through the steps.

First, what is the immediate successor in the in-order traversal?

This is the smallest value at the right subtree.

When we use in-order traversal, we visit the tree nodes in the ascending order.

A successor is a node that is visited after the node to be deleted. The immediate successor is the first node to visit after visiting the node to be deleted.

For node 27, the immediate successor is node 29. It is the smallest among all the nodes in the right subtree.

Do we have to use the immediate successor? Can we use the immediate predecessor? Can we use the largest value among the nodes in the left subtree? The answer is yes. We can use the immediate predecessor.

We can use either the immediate successor or the immediate predecessor. Either is acceptable.

In E cee E 2, 6, 4, we make a decision to use the immediate successor so that all students’ programs produce the same result.

The immediate successor is the left most child in the right subtree. It is important to note that the immediate successor must not have the left child. If this node has the left child, it is no longer the immediate successor.

After finding the immediate successor, we will swap the values stored in the two nodes.

Please notice that at this moment, this is not a binary search tree because 27 is at the right side of 29.

Fortunately, we will delete 27 soon. In the right subtree of 29, 27 is smaller than 29. All the other nodes are greater than 29.

Next, we are going to delete 27 in the right subtree. As mentioned earlier, the immediate successor must not have the left child. Thus, this will fall into the first case when the deleted node has no child or the second case when the deleted node has only one child.

In this example, after swapping, node 27 has no child. Thus, we will set the parent’s pointer to NULL and free the memory.

Let’s see how the delete function can be implemented in a cee program.

The delete function takes two arguments. The first argument is a pointer of tree node. The second argument is a value. We want to delete the node whose value is the same as the second argument.

The function returns the pointer of the tree after deleting the node. If this value does not existing in the tree, the function returns the original tree.

At the top of this function, as usually, we check whether this tree is empty or not. If the tree is empty, nothing can be deleted.

If the tree is not empty, we compare vee ei L with Tee N arrow value. If vee ei L is smaller, we will check whether vee ei L is in the left subtree.

This part will return Tee N because it is not possible to find vee ei L anywhere else and the rest of this delete function is not used.

If vee ei L is greater, we will check whether vee ei L is in the right subtree.

Please notice that this is symmetric with the earlier part that checks whether vee ei L is smaller.

If none of the condition is satisfied, then, vee ei L must be equal to this node’s value. We need to delete this node.

We need to check the three conditions: does this node have no child, one child, or two children.

The first case is when this node has no child. In that case, tee N arrow left is NULL. Tee N arrow right is also NULL.

The function free the memory of this node and returns NULL.

The second case is when this node has only one child. This only child can be the right child or the left child.

If the node’s left is NULL, this node has only the right child. Then, we keep the right child, free this node, and return the right child.

If the node’s right is NULL, this node has only the left child. Then, we keep the left child, free this node, and return the left child.

Please notice the symmetry of these two cases for left child or right child.

We can check either case first.

If none of the if condition is satisfied, then this node must have both children. We need to handle this case in a different way.

As explained earlier, we want to find the immediate successor. The program assigns the right child to ass you.

We already know that ass you must not be NULL because this has already been checked earlier.

Then, the program enters while to go as much left as possible. When the program gets out of while, ass you is the immediate successor.

After finding the immediate successor, we will swap the values of the two nodes.

Then, we will delete the value in the right subtree.

Finally, this function returns tee N. .

Please notice that tee N’s value is the value of the immediate successor since we have already swapped.

Let’s review some common mistakes.

One of the common mistakes is not checking whether this tree is empty or not. Checking this necessary because we need to check tee N arrow value in the next condition.

The next common mistake is not using the correct pointers. When vee ei L is smaller than Tee N arrow value, we need to check whether vee ei L is in the left subtree. Thus, the first argument of calling the delete function must be tee N arrow left. If the first argument is Tee N, not Tee N arrow left, then this recursive function is not making progress and recursion will not end.

For the pointer in the left side of the assignment, it must be tee N arrow left. It cannot be tee N. .

We already know that vee ei L is different from Tee N arrow value. That is the reason why the program enters this block of code.

If we put tee N at the left side of the assignment, then we will lose this node.

Also, please remember to return this node Tee N. .

Since this node is not deleted, It should be returned so that the parent node can still point to this node.

The next common mistake is forgetting to return NULL if this node has no child and this node is deleted. It is necessary returning NULL so that the parent node’s link can point to NULL.

Another common mistake is the order when this node has only one child. These three lines must follow this order.

First, we must have a pointer that keeps this node’s only child. Then, the program frees the memory of this node. The third step returns the child.

Let’s consider what may happen if the order is changed.

If we move return are Cee earlier, the function stops without freeing the memory. Thus, this is wrong.

If we move free tee N earlier, then we can no longer get Tee N arrow right. Thus, we must not move free Tee N earlier.

When we need to find the immediate successor, we must start from the right subtree. Thus, ass you must be the right child of Tee N. .

Ass you must not be equal to Tee N. .

The condition for while is ass you arrow left is not NULL. If you write ass you is not NULL, then when the program gets out of while, ass you is NULL and it is not a valid node.

The next common mistake is not using tee N arrow right in calling the delete function again. This is a similar problem explained earlier. Please notice that the values of tee N and its immediate successor have already been swapped.

Thus, we are no longer deleting this node, tee N. .

Instead, we want to delete the node that was earlier storing the value of the immediate successor.

Please study the delete function. It is not long but it is subtle in several places and mistakes may happen.