## Insertion

Insertion can be done in multiple ways depending on the location where the element is to be inserted.

- We can insert the new element at the rightmost or the leftmost vacant position
- Or just insert the element in the first vacant position we find when we traverse a tree

As we said there are many more ways we can insert a new element, but for this article's sake let's try the first method:
Let's determine the vacant place with In-order traversal.
Parameters: **root, new_node**

1. Check the root if it's null, i.e., if it's an empty tree. If yes, return the new_node as root.
2. If it’s not, start the inorder traversal of the tree
3. Check for the existence of the left child. If it doesn't exist, the new_node will be made the left child, or else we'll proceed with the inorder traversal to find the vacant spot
4. Check for the existence of the right child. If it doesn't exist, we'll make the right child as the new_node or else we'll continue with the inorder traversal of the right child.

Here’s the Pseudo-code to run this operation-

```
TreeNode insert_node_inorder(TreeNode root, TreeNode new_node)

{

   if ( root == NULL )

      return new_node
```

```
    if ( root.left == NULL )
        root.left = new_node
    else
        root.left = insert_node_inorder(root.left, new_node)
    if ( root.right == NULL )
        root.right = new_node
    else
        root.right = insert_node_inorder(root.right, new_node)

    return root
}
```

**Searching**

It's a simple process in a binary tree. We just need to check if the current node's value matches the required value and keep repeating the same process to the left and right subtrees using a recursive algorithm until we find the match.

```
bool search(TreeNode root, int item)
{
    if ( root == NULL )
        return 0
    if ( root.val == item )
        return 1
```

```
    if ( search(root.left, item) == 1 )
        return 1
    else if ( search(root.right, item) == 1 )
        return 1
    else
        return 0
}
```

**Deletion**

It's a bit tricky process when it comes to the tree data structure. There are a few complications that come with deleting a node such as-

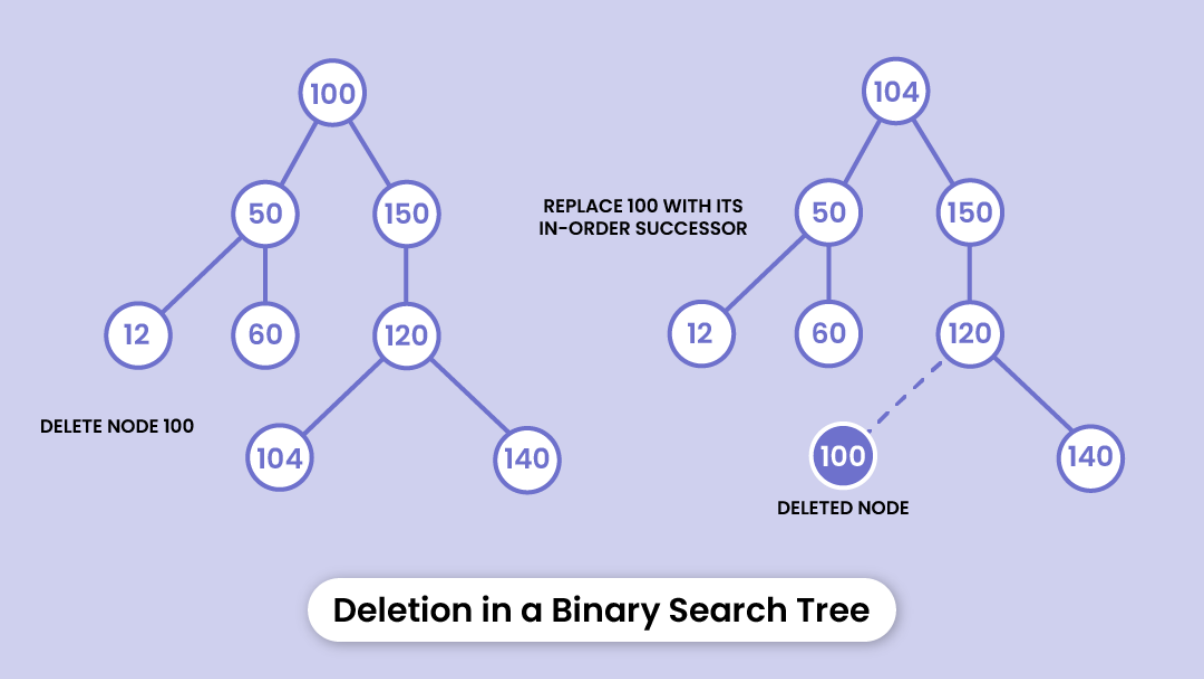
- If we delete a node, what happens to the left child and the right child?
- What if the node to be deleted is itself a leaf node?

Simplifying this - the purpose is to accept the root node of the tree and value item and return the root of the modified tree after we have deleted the node.

- Firstly, we'll check if the tree is empty i.e. the root is NULL. If yes, we'll simply return the root.
- We'll then search for an item in the left and the right subtree and recurse if found.
- If we don't find the item in both the subtrees, either the value is not in the tree or root.val == item.

- Now we need to delete the root node of the tree. It has three possible cases.

**CASE 1** - The node to be deleted is a leaf node.
In this case, we'll simply delete the root and free the allocated space.
**CASE 2** - It has only one child.
We can't delete the root directly as there's a child node attached to it. So, we'll replace the root with the child node.
**CASE 3** - It has two children nodes.
In this case, we'll keep replacing the node to be deleted with its in-order successor recursively until it's placed on the leftmost leaf node. Then, we'll replace the node with NULL and delete the allocated space.
In other words, we'll replace the node to be deleted with the leftmost node of the tree and then delete the new leaf node.
This way, there's always a root at the top and the tree shrinks from the bottom.




Here’s the pseudo-code to execute a deletion:

```
TreeNode delete_element(TreeNode root, int item)
{
   if ( root == NULL )
      return root

   if ( search(root.left, item) == True )
      root.left = delete_element(root.left, item)
   else if ( search(root.right, item) == True )
      root.right = delete_element(root.right, item)

   else if ( root.val == item )
   {
      // No child exists
      if ( root.left == NULL and root.right == NULL )
         delete root
      // Only one child exists
      else if ( root.left == NULL or root.right == NULL )
      {
         if ( root.left == NULL )
            return root.right
         else
```

```
            return root.left
        }
        // Both left and right child exists
        else
        {
            TreeNode selected_node = root
            while ( selected_node.left != NULL )
                selected_node = selected_node.left
            root.val = selected_node.val
            root.left = delete_element(root.left, selected_node.val)
        }
    }
    return root
}
```

**Applications of Tree Data Structure**

As we've mentioned above, tree data structure stores data in a hierarchical manner. Nodes are arranged at multiple levels.

- Information stored in the computer is in a hierarchical manner. There are drives that contain multiple folders. Each folder can have multiple subfolders. And then there are files like documents, images, etc.