Sure, let's walk through the process flow step by step to understand how the replacement of the target node with its child's data occurs during the node deletion in the AVL tree. We'll consider the case where the target node has only one child (either the left child or the right child).

1. **Identifying the Node to Delete:**
   * The **deleteNode** function starts by recursively traversing the AVL tree to find the node that needs to be deleted based on the **nodeValue** provided.
2. **Node with One Child (Left Child is None):**
   * Once the target node is found and it is determined that it has only one child (the left child is **None**), the code enters the **if rootNode.leftChild is None** block.
   * In this case, the target node has a right child (could be a leaf node or an internal node).
3. **Storing the Right Child in a Temporary Variable (temp):**
   * The code creates a temporary variable **temp** and assigns it the value of the right child of the target node (if it exists).
   * Since the left child is **None**, **temp** now holds the right child (which could be either **None** or an actual node).
4. **Deleting the Target Node:**
   * The code sets the target node (**rootNode**) to **None**, effectively deleting the target node from the AVL tree.
5. **Returning the Temporary Variable (temp):**
   * Finally, the code returns the value stored in the temporary variable **temp**.
   * This is crucial for the recursive nature of the function call. When this function call returns, the parent node's left or right child (depending on the original target node's position) will be updated to the value returned by this function call.
   * So, effectively, the target node is replaced by its child in the parent node.
6. **Node with One Child (Right Child is None):**
   * If the target node has only one child and the right child is **None**, the code enters the **elif rootNode.rightChild is None** block.
   * In this case, the target node has a left child (which could be either **None** or an actual node).
7. **Storing the Left Child in a Temporary Variable (temp):**
   * Similar to the previous case, the code creates a temporary variable **temp** and assigns it the value of the left child of the target node (if it exists).
8. **Deleting the Target Node:**
   * The code sets the target node (**rootNode**) to **None**, effectively deleting the target node from the AVL tree.
9. **Returning the Temporary Variable (temp):**
   * The code returns the value stored in the temporary variable **temp**.
   * When this function call returns, the parent node's left or right child (depending on the original target node's position) will be updated to the value returned by this function call.
   * The target node is replaced by its child in the parent node.

In both cases, the child of the target node (either the left child or the right child, depending on the scenario) effectively takes the place of the target node in the parent node. This is achieved by updating the parent node's left or right child pointer to point to the child node instead of the original target node.

The recursive nature of the function call ensures that the changes are propagated up the tree, and any parent nodes pointing to the target node will have their child pointer updated accordingly. As a result, the AVL tree structure is maintained after deleting the target node.