BINARY SEARCH TREE DELETION ALGORITHM

```
// function to find minimum value node in the subtree rooted at `curr`

Node* getMinimumKey(Node* curr)
{
    while (curr->left != nullptr) {
        curr = curr->left;
    }
    return curr;
}

// Function to search in the subtree rooted at `curr` and set its parent.
//  `curr` and `parent` is passed by reference to the function.
void searchKey(Node* &curr, int key, Node* &parent)
{
    // traverse the tree and search for the key
    while (curr != nullptr && curr->data != key)
    {
        // update the parent to the current node
        parent = curr;
         // if the given key is less than the current node, go to the left
           subtree; otherwise, go to the right subtree
        if (key < curr->data) {
            curr = curr->left;
        }
        else {
            curr = curr->right;
        }
    }
}

void deleteNode(Node*& root, int key)
{
    // pointer to store the parent of the current node
    Node* parent = nullptr;
     // start with the root node
       Node* curr = root;
     // search key in the BST and set its parent pointer

   searchKey(curr, key, parent);
    // return if the key is not found in the tree
    if (curr == nullptr) {
        return;
    }

    // Case 1: node to be deleted has no children, i.e., it is a leaf node
    if (curr->left == nullptr && curr->right == nullptr)
```

```
{
    // if the node to be deleted is not a root node, then set its
    // parent left/right child to null
    if (curr != root)
    {
        if (parent->left == curr) {
            parent->left = nullptr;
        }
        else {
            parent->right = nullptr;
        }
    }
    // if the tree has only a root node, set it to null
    else {
        root = nullptr;
    }

    // deallocate the memory
    free(curr);        // or delete curr;
}

// Case 2: node to be deleted has two children
else if (curr->left && curr->right)
{
    // find its inorder successor node
    Node* successor = getMinimumKey(curr->right);

    // store successor value
    int val = successor->data;

    // recursively delete the successor. Note that the successor
    // will have at most one child (right child)
    deleteNode(root, successor->data);

    // copy value of the successor to the current node
    curr->data = val;
}

// Case 3: node to be deleted has only one child
else {
    // choose a child node
    Node* child = (curr->left)? curr->left: curr->right;

    // if the node to be deleted is not a root node, set its parent
    // to its child
    if (curr != root)
    {
        if (curr == parent->left) {
            parent->left = child;
        }
```

```
            else {
                parent->right = child;
            }
        }

        // if the node to be deleted is a root node, then set the root to the child
        else {
            root = child;
        }

        // deallocate the memory
        free(curr);
    }
}
```