Binary search tree

```
#include <bits/stdc++.h>
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
};
// create a new BST node
struct TreeNode* New(int value) {
    struct TreeNode* temp = (struct TreeNode*)calloc(1, sizeof(struct TreeNode));
    temp->val = value;
    temp->left = temp->right = nullptr;
    return temp;
}
struct TreeNode* Insert(struct TreeNode* node, int value) {
    // if the BST is empty, return a new node
    if (node == nullptr) {
        return New(value);
    }
    // otherwise, recur down the tree
    if (value < node->val) {
        node->left = Insert(node->left, value);
    }
    else if (value >= node->val) {
        node->right = Insert(node->right, value);
    }
    // return the (unchanged) node pointer
    return node;
}
// display the BST in a inorder traversal
void DisplayTree(struct TreeNode* node) {
    if (node != nullptr) {
        DisplayTree(node->left);
        std::cout << node->val << std::endl;
        DisplayTree(node->right);
    }
}
bool SearchTree(struct TreeNode* node, int key) {
    if (node == nullptr) {
        std::cout << "Couldn't find element! Reached NULL!" << std::endl;
        return false;
    }
```

```
    else if (node != nullptr) {
        if (key == node->val) {
            std::cout << "Key of " << node->val << " is Found!" << std::endl;
            return true;
        }
        std::cout << "Current node value is: " << node->val << std::endl;
        if (key < node->val) {
            std::cout << key << " is smaller! Going left!" << std::endl;
            return SearchTree(node->left, key);
        }
        else if (key > node->val) {
            std::cout << key << " is larger! Going right!" << std::endl;
            return SearchTree(node->right, key);
        }
    }
    std::cout << "something went wrong!!" << std::endl;
    return false;
}
struct TreeNode* minValNode(struct TreeNode* node) {
    struct TreeNode* current = node;
    // loop down to find the leftmost leaf
    while (current && current->left != nullptr) {
        current = current->left;
    }
    return current;
}
struct TreeNode* DeleteNode1(struct TreeNode* root, int key) {
    if (root == nullptr) {
        return root;
    }
    if (key < root->val) {
        // std::cout << key << " is smaller! Going left!" << std::endl;
        root->left = DeleteNode1(root->left, key);
    }
    else if (key > root->val) {
        // std::cout << key << " is larger! Going right!" << std::endl;
        root->right = DeleteNode1(root->right, key);
    }
    else if (key == root->val) {
        // std::cout << "Found value!" << std::endl;
        // node has no child
        if (root->left == nullptr && root->right == nullptr) {
```

```
            return nullptr;
        }
        // node with only one child or no child
        else if (root->left == nullptr) {
            struct TreeNode* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == nullptr) {
            struct TreeNode* temp = root->left;
            free(root);
            return temp;
        }
        // node with two children: Get the inorder successor
        // (smallest in the right subtree)
        struct TreeNode* temp = minValNode(root->right);
        // Copy the inorder successor's content to this node
        root->val = temp->val;
        // Delete the inorder successor
        root->right = DeleteNode1(root->right, temp->val);
    }
    return root;
}
struct TreeNode* del2(TreeNode* node) {
    if (node == nullptr) {
        return nullptr;
    }
    if (node->right == nullptr) {
        // free(node);
        return node->left;
    }
    struct TreeNode* temp = node->right;
    while (temp->left) {
        temp = temp->left;
    }
    temp->left = node->left;
    return node->right;
}
struct TreeNode* DeleteNode2(struct TreeNode* root, int key) {
    if (root == nullptr) {
        return nullptr;
    }
```

```
    struct TreeNode* curr = root;
    struct TreeNode* prev = nullptr;
    while (curr != nullptr) {
        if (curr->val == key) break;
        prev = curr;
        if (curr->val > key) {
            curr = curr->left;
        }
        else if (curr->val < key) {
            curr = curr->right;
        }
    }
    if (prev == nullptr) {
        return del2(curr);
    }
    if (prev->left != nullptr && prev->left->val == key) {
        prev->left = del2(curr);
    }
    else if (prev->left == nullptr || prev->left->val != key) {
        prev->right = del2(curr);
    }
    return root;
}
struct TreeNode* DeleteNode3(struct TreeNode* root, int key) {
    if (root == nullptr) {
        return nullptr;
    }
    if (root->val == key) {
        if (root->right == nullptr) {
            return root->left;
        }
        else if (root->right != nullptr) {
            struct TreeNode *current = root->right;
            while (current->left != nullptr) {
                current = current->left;
            }
            std::swap(root->val, current->val);
        }
    }
    root->left = DeleteNode3(root->left, key);
    root->right = DeleteNode3(root->right, key);
    return root;
```

```
}
int main() {
    std::vector<int> vals = {90, 80, 70, 60, 50, 40, 30, 20, 10};
    struct TreeNode* root = nullptr;
    for (int i = 0; i < vals.size(); i++) {
        root = Insert(root, vals[i]);
    }
    std::cout << std::endl;

    // show that this is actually a BST
    std::cout << "----------Process of searching 20----------\n";
    SearchTree(root, 20);
    std::cout << "----------Process of searching 40----------\n";
    SearchTree(root, 60);
    std::cout << "----------Process of searching 80----------\n";
    SearchTree(root, 80);

    std::cout << "----------Inorder traversal of the original tree----------\n";
    DisplayTree(root);

    // three different delete methods
    root = DeleteNode1(root, 10);
    std::cout << "----------Inorder traversal after delete 10----------\n";
    DisplayTree(root);

    root = DeleteNode2(root, 70);
    std::cout << "----------Inorder traversal after delete 70----------\n";
    DisplayTree(root);

    root = DeleteNode3(root, 90);
    std::cout << "----------Inorder traversal after delete 90----------\n";
    DisplayTree(root);

    std::cout << std::endl;
    return 0;
}
```