BINARY SEARCH TREE SAMPLE CODE
==============================

```
struct node
{
    int key;
    node *left = nullptr;
    node *right = nullptr;
};

class BST
{
public:

    bool add(int newkey);
    bool remove(int removekey);
    bool contains(int searchkey) const;

private:
    node *root = nullptr;

    bool add(node *curr, int newkey);
    bool contains(node *curr, int searchkey) const;
};

bool BST::add(int newkey)
{
    if (root != nullptr)
        return add(root, newkey);
    else
    {
        root = new node;
        root->key = newkey;
        return true;
    }
}

bool BST::add(node *curr, int newkey)
{
    if (curr->key == newkey)
        return false; // key is already in BST
    else if (newkey < curr->key)
    {
        if (curr->left == nullptr)
        {
            curr->left = new node;
            curr->left->key = newkey;
            return true;
        }
        else
            return add(curr->left, newkey);
    }
    else
    {
        if (curr->right == nullptr)
        {
            curr->right = new node;
            curr->right->key = newkey;
            return true;
        }
        else
            return add(curr->right, newkey);
    }
}
```

```
bool BST::contains(int searchkey) const
{
    return contains(root, searchkey);
}

bool BST::contains(node *curr, int searchkey) const
{
    if (curr == nullptr)    // key not found
        return false;
    else if (searchkey == curr->key)
        return true;        // key found
    else if (searchkey < curr->key)
        return contains(curr->left, searchkey);
    else
        return contains(curr->right, searchkey);
}

bool BST::remove(int removekey)
{
    node *curr = root;          // Node that will be deleted.
    node *parent = nullptr;     // Parent of node that will be deleted (or null if deleting the root).
    while (curr != nullptr && curr->key != removekey)
    {
        // Descend through the tree, looking for the node that contains removekey.
        // Stop when we find it, or when we encounter a null pointer.
        parent = curr;
        if (removekey < curr->key)
            curr = curr->left;
        else
            curr = curr->right;
    }
    // At this point, curr is null, or we've found removekey.
    if (curr == nullptr)
        return false; // removekey was not in the tree

    // We've found removekey in the "curr" node, so delete curr from the tree.
    if (curr->left != nullptr && curr->right != nullptr)  // Handle 2-child situation first.
    {
        node *successor = curr->right;  // Find inorder successor (minimum element in right subtree).
        node *successorParent = curr;
        while (successor->left != nullptr)
        {
            successorParent = successor;
            successor = successor->left;
        }
        // Copy the successor's key into curr.
        curr->key = successor->key;
        // Continue with code below that will delete the successor node (guaranteed to have < 2 children).
        curr = successor;
        parent = successorParent;
    }

    // Handle if curr has zero or one child.
    node *subtree;  // Pointer to the subtree of curr that exists, if there is one, or null if it has 0 children.
    if (curr->left == nullptr && curr->right == nullptr)    // No children.
        subtree = nullptr;
    else if (curr->left != nullptr)     // Only a left child.
        subtree = curr->left;
    else
        subtree = curr->right;          // Only a right child.

    // Attach subtree to the correct child pointer of the parent node, if it exists.
    // If there is no parent, then we are deleting the root node, and the subtree becomes the new root.
    if (parent == nullptr)
        root = subtree;
    else if (parent->left == curr)  // Deleting parent's left child.
        parent->left = subtree;
    else
        parent->right = subtree;    // Deleting parent's right child.

    delete curr;

    return true; // successful deletion
}
```