```
class Node
{
    int key;
    Node left = null;
    Node right = null;
}

public class BST {
    private Node root = null;

    /**
     * Add a new key into this BST.  Returns true for a successful add (when the key wasn't already in
     * the BST), and false if unsuccessful (when the key was already there).
     */
    public boolean add(int newKey)
    {
        if (root == null) // tree is empty
        {
            Node newNode = new Node();
            newNode.key = newKey;
            root = newNode;
            return true; // successful add
        }
        else
            return add(root, newKey);
    }

    private boolean add(Node curr, int newKey)
    {
        if (curr.key == newKey) // newKey already exists, can't add it twice
            return false;

        else if (newKey < curr.key) // key should be inserted in the left branch
        {
            if (curr.left == null)  // is the left branch empty?
            {
                Node newNode = new Node();
                newNode.key = newKey;
                curr.left = newNode;
                return true; // successful add
            }
            else
                return add(curr.left, newKey);
        }
        else  // key should be inserted in the right branch
        {
            if (curr.right == null)  // is the right branch empty?
            {
                Node newNode = new Node();
                newNode.key = newKey;
                curr.right = newNode;
                return true; // successful add
            }
            else
                return add(curr.right, newKey);
        }
    }

    /**
     * Return true if this BST contains searchKey, false otherwise.
     */
    public boolean contains(int searchKey)
    {
        return contains(root, searchKey);
    }

    private boolean contains(Node curr, int searchKey)
    {
        if (curr == null)
            return false;       // reached a leaf node, not found -> key not in tree

        else if (searchKey == curr.key)
            return true;        // key found

        else if (searchKey < curr.key) // searchKey too small -> go left
            return contains(curr.left, searchKey);

        else                           // searchKey too big -> go right
            return contains(curr.right, searchKey);
    }
```

```java
/**
 * Remove a key from the BST.  Returns true for a successful removal (when the key was in the tree
 * and has been removed), or false if the key wasn't in the BST in the first place.
 */
public boolean remove(int removeKey)
{
    Node curr = root; // Will point to the node to be deleted.
    Node parent = null; // Will point to the parent of the node to be deleted.
    while (curr != null && 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 successfully found removeKey in the tree.
    if (curr == null)
        return false; // removeKey wasn't found in the tree.

    // We've found removeKey at the "curr" node, so remaining code will
    // delete curr from the tree.

    // Handle 2-child situation first.
    if (curr.left != null && curr.right != null)
    {
        // Find inorder successor of curr.
        Node successor = curr.right;
        Node successorParent = curr;
        while (successor.left != null)
        {
            successorParent = successor;
            successor = successor.left;
        }
        // Copy the inorder successor's key into curr.
        curr.key = successor.key;
        // Continue with the code below that will delete the successor node, which
        // is guaranteed to have 0 children or 1 child.
        curr = successor;
        parent = successorParent;
    }

    // Handle 0-child or 1-child situation.
    Node subtree; // Will point to the subtree of curr that exists, if there is one,
    // or null if it has 0 children.

    if (curr.left == null && curr.right == null) // 0 children
        subtree = null;
    else if (curr.left != null)
        subtree = curr.left;
    else
        subtree = curr.right;

    // 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 == null)
        root = subtree;
    else if (parent.left == curr)   // Deleting parent's left child.
        parent.left = subtree;
    else                            // Deleting parent's left child.
        parent.right = subtree;

    return true; // successful deletion.
}
```