```
class BinaryTreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
class BST:

    def __init__(self):
        self.root = None
        self.numNodes = 0

    #Time complexity: O(H) [for all operations]
    #Space complexity: O(N)
    #
    #where N is the number of nodes in the input BST
    #and H is the height of the input BST

    def printTreeHelper(self, root):
        if root == None:
            return
        print(root.data, end = ":")
        if root.left != None:
            print("L:",end='')
            print(root.left.data,end=',')
        if root.right != None:
            print("R:",end='')
            print(root.right.data,end='')
        print()
        self.printTreeHelper(root.left)
        self.printTreeHelper(root.right)

    def printTree(self):
        self.printTreeHelper(self.root)

    def isPresentHelper(self, root, data):
        if root == None:
            return False

        if root.data == data:
            return True

        if root.data > data:
            # call on left
            return self.isPresentHelper(root.left, data)
        else:
            # call on right
            return self.isPresentHelper(root.right, data)

    def search(self, data):
        return self.isPresentHelper(self.root, data)

    def insertHelper(self, root, data):
        if root == None:
            node = BinaryTreeNode(data)
            return node

        if root.data >= data:
            root.left = self.insertHelper(root.left, data)
            return root
        else:
            root.right = self.insertHelper(root.right, data)
            return root

    def insert(self, data):
        self.numNodes += 1
        self.root = self.insertHelper(self.root, data)

    def min(self, root):
        if root == None:
```

```
            return 10000

        if root.left == None:
            return root.data

        return self.min(root.left)

    def deleteDataHelper(self, root, data):
        if root == None:
            return False, None

        if root.data < data:
            deleted, newRightNode = self.deleteDataHelper(root.right, data)
            root.right = newRightNode
            return deleted, root

        if root.data > data:
            deleted, newLeftNode = self.deleteDataHelper(root.left, data)
            root.left = newLeftNode
            return deleted, root

        # root is leaf
        if root.left == None and root.right == None:
            return True, None

        # root has one child
        if root.left == None:
            return True, root.right

        if root.right == None:
            return True, root.left

        # root has two children
        replacement = self.min(root.right)
        root.data = replacement
        deleted, newRightNode = self.deleteDataHelper(root.right, replacement)
        root.right = newRightNode
        return True, root

    def delete(self, data):
        deleted, newRoot = self.deleteDataHelper(self.root, data)
        if deleted:
            self.numNodes -= 1
        self.root = newRoot
        return deleted

    def count(self):
        return self.numNodes

b = BST()
q = int(input())
while (q > 0) :
    li = [int(ele) for ele in input().strip().split()]
    choice = li[0]
    q-=1
    if choice == 1:
        data = li[1]
        b.insert(data)
    elif choice == 2:
        data = li[1]
        b.delete(data)
    elif choice == 3:
        data = li[1]
        ans = b.search(data)
        if ans is True:
            print('true')
        else:
            print('false')
```

```
    else:
        b.printTree()
```