#include <iostream>

using namespace std;

// A Node type for binary trees

struct Node {

int data;

Node \*left, \*right;

};

// Return a new node with the given data, left and right subtrees.

Node \* makeTree(int data, Node \*left = nullptr, Node \*right = nullptr) {

Node \* temp = new Node;

temp->left = left;

temp->right = right;

temp->data = data;

return temp;

}

// Return the smallest value in a nonempty BST with root r.

int getSmallest(Node \* r) {

// FILL IN MISSING CODE

return 0; // remove this line after filling in missing code

}

// Remove the node with smallest value from a nonempty BST

// and return a pointer to the updated tree.

// Usually return value will be r, unless smallest value is in r.

// In case smallest value is in r, return r->right.

// Make sure to call delete on the node that is removed.

Node \* removeSmallest(Node \* r) {

if (r->left == nullptr) {

Node \* temp = r;

r = r->right;

delete temp;

return r;

}

// FILL IN MISSING CODE

return nullptr; // REMOVE THIS AFTER FILLING IN MISSING CODE

}

// Return a pointer to the node that will be

// the parent of a new node containing v,

// or return nullptr if v is already in tree.

// This function is complete. There is no code to add.

Node \* findInsertionPoint(Node \*r, int v) {

if (r->data == v) return nullptr;

if (v < r->data) {

if (r->left == nullptr) return r;

return findInsertionPoint(r->left, v);

}

else {

if (r->right == nullptr) return r;

return findInsertionPoint(r->right, v);

}

}

// Return true if the BST contains v and false otherwise.

bool contains(Node \* r, int v) {

if (r == nullptr) return false;

// FILL IN MISSING CODE

return true; // remove this after you fill in missing code

}

// Insert v into a nonempty binary search tree.

// Call findInsertionPoint to get a pointer to

// the node below which v should be inserted.

// If findInsertionPoint returns nullptr, don't do anything

// since that means v is already in the tree.

void insertValue(Node \* r, int v) {

Node \* p = findInsertionPoint(r, v);

// FILL IN MISSING CODE

}

// Output the values in the BST using an inorder traversal.

void inorder(Node \* r) {

// FILL IN MISSING CODE

}

// Print the binary tree.

// This function is complete. There is no code to add.

void printTree(Node \* r, int offset = 4) {

char c = r == nullptr ? '.' : '-';

for (int i = 0; i < offset - 4; i++)

cout << ' ';

for (int i = 0; i < 4; i++)

cout << c;

if (r != nullptr) {

cout << r->data << endl;

offset += 4;

printTree(r->left, offset);

printTree(r->right, offset);

}

else cout << '.' << endl;

}

// DON'T ALTER THE CODE BELOW UNLESS TO COMMENT OUT SOME TESTS

int main() {

Node \*r = makeTree(7);

printTree(r);

bool quit = false;

do {

cout << endl << "Enter new value (or -1 to quit): ";

int v;

cin >> v;

if (v != -1) {

cout << "Before insertion, contains(r," << v << ") = " << contains(r, v) << "." << endl;

insertValue(r, v);

cout << "After insertion, contains(r," << v << ") = " << contains(r, v) << "." << endl;

cout << "Updated tree = " << endl;

printTree(r);

}

else quit = true;

} while (!quit);

cout << endl << "inorder traversal = { ";

inorder(r);

cout << "}" << endl;

while (r != nullptr) {

cout << "removing " << getSmallest(r) << endl;

r = removeSmallest(r);

printTree(r);

}

cout << endl << "DONE" << endl;

return 0;

}