```c
#include <stdbool.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <limits.h>

typedef struct NODE Node;

//============================================================================
//Stack
//============================================================================

//The stack is used to remember the unvisited nodes when traversing the tree iteratively
//A dynamic array would've also been possible, but the stack is easier to use
typedef struct STACK_ELEMENT StackElement;

struct STACK_ELEMENT {
    Node* value;
    StackElement *next;
};

typedef struct Stack {
    StackElement* top;
} Stack;

Stack stack_create() {
    Stack stack;
    stack.top = NULL;

    return stack;
}

StackElement* stack_create_node(Node* value) {
    StackElement* element = (StackElement*)malloc(sizeof(StackElement));
    element->value = value;
    element->next = NULL;

    return element;
}

void stack_push(Stack* stack, Node* value) {
    if(stack->top == NULL) {
        stack->top = stack_create_node(value);
        return;
    }

    StackElement* element = stack_create_node(value);
    element->next = stack->top;
    stack->top = element;
}

Node* stack_pop(Stack* stack) {
    Node* value = stack->top->value;
    StackElement* top = stack->top;
    stack->top = stack->top->next;
    free(top);

    return value;
}

bool stack_empty(Stack stack) {
    return stack.top == NULL;
}

void stack_delete(Stack* stack) {
    while(!stack_empty(*stack)) {
        stack_pop(stack);
    }
}
```

```
//================================================================================
//Tree
//================================================================================

struct NODE {
    int key;
    Node* smaller_keys;
    Node* larger_keys;
};

typedef struct {
    Node* root;
} Tree;

Tree tree_create() {
    Tree tree;
    tree.root = NULL;

    return tree;
}

Node* node_create(int key) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->key = key;
    node->smaller_keys = NULL;
    node->larger_keys = NULL;

    return node;
}

//Finds a node with a given key recursively
//Mechanism: Look at the current node. If the key is smaller than the node's
//key, go left if possible or return the node. Do the same for bigger and going right.
Node* tree_get_candidate_node(Node *node, int key) {
    if(node->smaller_keys && key < node->key) {
        return tree_get_candidate_node(node->smaller_keys, key);
    }
    else if(node->larger_keys && key > node->key) {
        return tree_get_candidate_node(node->larger_keys, key);
    }
    else {
        return node;
    }
}

bool tree_find_key_recursive(Tree tree, int key) {
    return tree_get_candidate_node(tree.root, key)->key == key;
}

Node* tree_insert_key(Tree* tree, int key) {
    Node* new_node = node_create(key);
    if(!tree->root) {
        tree->root = new_node;
        return new_node;
    }

    Node* current = tree->root;
    while(current) {
        if(key < current->key) {
            if(!current->smaller_keys) {
                current->smaller_keys = new_node;
                return new_node;
            }
            current = current->smaller_keys;
        }
        else if(key > current->key) {
            if(!current->larger_keys) {
                current->larger_keys = new_node;
                return new_node;
            }
```

```
            current = current->larger_keys;
        }
        else {
            free(new_node);
            return NULL; //Ignore duplicates
        }
    }

    return NULL;
}

//tree_deep_copy, tree_delete and tree_is_valid use iterative traversion of the tree.
//To avoid code duplication, the traversal part is taken care of by the
//TreeTraverser struct
typedef struct {
    Node* current;
    Stack left_turns;
    Stack right_turns;
} TreeTraverser;

TreeTraverser tree_traverser_create(Node* node) {
    TreeTraverser t = {
        .current = node,
        .left_turns = stack_create(),
        .right_turns = stack_create()
    };

    return t;
}

void tree_traverser_delete(TreeTraverser* tree_traverser) {
    stack_delete(&tree_traverser->left_turns);
    stack_delete(&tree_traverser->right_turns);
}

//Traversing a tree iteratively: If the current node has left or right children,
//add them to the stacks of nodes to be visited. If the stacks contain nodes, take
//one from the top. Otherwise, set the current node to NULL, indicating that
//traversal is over.
void move_to_next_level(TreeTraverser* t) {
    if(t->current->smaller_keys) {
        stack_push(&t->left_turns, t->current->smaller_keys);
    }
    if(t->current->larger_keys) {
        stack_push(&t->right_turns, t->current->larger_keys);
    }

    if(!stack_empty(t->left_turns)) {
        t->current = stack_pop(&t->left_turns);
    }
    else if(!stack_empty(t->right_turns)) {
        t->current = stack_pop(&t->right_turns);
    }
    else {
        t->current = NULL;
    }
}

bool tree_traverser_end_reached(TreeTraverser t) {
    return t.current == NULL;
}

Node* tree_find_key_iterative(Tree tree, int key) {
    Node* current = tree.root;

    while(current) {
        if(current->key == key) {
            return current;
        }
        if(key > current->key) {
```

```
            current = current->larger_keys;
        }
        else {
            current = current->smaller_keys;
        }
    }

    return NULL;
}

/*Checks if a subtree of a node is valid. Mechanism:
1. Get the maximum key of the subtree to the left.
2. If the maximum is greater or equal to the node's value, the tree is invalid.
Same for right subtree, but reversed.
*/
bool _is_valid(Node node) {
    if(!node.smaller_keys && !node.larger_keys) {
        return true;
    }

    if(node.smaller_keys) {
        int left_min = INT_MAX;
        Node* current = node.smaller_keys;
        while(current) {
            if(current->key < left_min) {
                left_min = current->key;
            }
            current = current->smaller_keys;
        }
        if(left_min >= node.key) {
            return false;
        }
    }
    if(node.larger_keys) {
        int right_max = INT_MIN;
        Node* current = node.larger_keys;
        while(current) {
            if(current->key > right_max) {
                right_max = current->key;
            }
            current = current->larger_keys;
        }
        if(right_max <= node.key) {
            return false;
        }
    }

    return true;
}

bool tree_is_valid(Tree tree) {
    TreeTraverser t = tree_traverser_create(tree.root);
    while(!tree_traverser_end_reached(t)) {
        if(!_is_valid(*t.current)) {
            tree_traverser_delete(&t);
            return false;
        }
        move_to_next_level(&t);
    }

    return true;
}

//The iterative traversal uses stacks to remember which nodes to visit.
//This causes the keys to automatically be in the right order for deep copying
//the tree structure.
Tree tree_deep_copy(Tree tree) {
    Tree copy = tree_create();

    TreeTraverser t = tree_traverser_create(tree.root);
```

```
    while(!tree_traverser_end_reached(t)) {
        tree_insert_key(&copy, t.current->key);
        move_to_next_level(&t);
    }

    return copy;
}

void tree_delete(Tree* tree) {
    TreeTraverser t = tree_traverser_create(tree->root);
    while(!tree_traverser_end_reached(t)) {
        Node* to_delete = t.current;
        move_to_next_level(&t);
        free(to_delete);
    }
    tree->root = NULL;
}

//============================================================================
//Testing
//============================================================================

void test_tree_is_valid() {
    Tree tree = tree_create();
    tree.root = node_create(0);
    tree.root->larger_keys = node_create(-1);
    tree.root->smaller_keys = node_create(15);

    assert(!tree_is_valid(tree));
}

void test_deep_copy() {
    Tree tree = tree_create();
    tree.root = node_create(7);
    tree.root->larger_keys = node_create(10);
    tree.root->smaller_keys = node_create(2);
    tree.root->smaller_keys->larger_keys = node_create(5);
    tree.root->smaller_keys->larger_keys->smaller_keys = node_create(3);
    tree.root->larger_keys->larger_keys = node_create(15);

    Tree copy = tree_deep_copy(tree);

    assert(tree.root->key == 7);
    assert(tree.root->larger_keys->key == 10);
    assert(tree.root->smaller_keys->key == 2);
    assert(tree.root->smaller_keys->larger_keys->key == 5);
    assert(tree.root->smaller_keys->larger_keys->smaller_keys->key == 3);
    assert(tree.root->larger_keys->larger_keys->key == 15);

    tree_delete(&copy);
    tree_delete(&tree);
}

void test_insertion_and_deletion() {
    Tree tree = tree_create();
    assert(tree_is_valid(tree));

    const int SIZE = 100;

    Node* nodes[SIZE];
    for(int i = 0; i < SIZE; ++i) {
        int value = rand() % (SIZE*100);
        nodes[i] = tree_insert_key(&tree, value);
        if(nodes[i] == NULL) { //Duplicate entry
            --i; //Try again
        }
    }
    assert(tree_is_valid(tree));

    //Test finding keys iteratively and recursively
```

```
    Tree copy = tree_deep_copy(tree);
    for(int i = 0; i < SIZE; ++i) {
        assert(tree_find_key_iterative(tree, nodes[i]->key));
        assert(tree_find_key_recursive(tree, nodes[i]->key));
        assert(tree_find_key_iterative(copy, nodes[i]->key));
    }

    tree_delete(&copy);
    for(int i = 0; i < SIZE; ++i) {
        assert(tree_find_key_iterative(tree, nodes[i]->key));
        assert(!tree_find_key_iterative(copy, nodes[i]->key));
    }
    tree_delete(&tree);
    assert(tree.root == NULL);
}

int main() {
    test_tree_is_valid();
    test_deep_copy();
    test_insertion_and_deletion();

    printf("All tests passed!\n");

    return 0;
}
```