L11 - Lecture - Binary Search Tree 2

BST as a full-fledged C++ class:

Interface (bst-2.h)

```
#ifndef __BST_2_H__
#define __BST_2_H__

#include <iostream>

using namespace std;

class BSTNode {

    public:
        int get() const { return data; }

        friend class BST;

    private:
        int data;
        BSTNode *left;
        BSTNode *right;

        BSTNode(int x) {
            data = x;
            left = NULL;
            right = NULL;
        }

        // disallow copy ctor & op=()
        BSTNode(const BSTNode&);
        BSTNode& operator=(const BSTNode&);
};

class BST {

    public:
        BST() : root(NULL) {}
        ~BST();
        BST(const BST&);
        BST& operator=(const BST&);

        void insert(int x);
        void remove(int x);

        typedef BSTNode Node;

        const Node *lookup(int x) const;

        void draw();
        void traverse_inorder(void (*f)(Node *));

        int height() const;
```

```
        const Node *find_min() const;
        const Node *find_max() const;

    private:
        Node *root;

        Node *insert(int x, Node *node);
        void remove(int x, Node*& node);
        const Node *lookup(int x, const Node *node) const;

        void remove_all_nodes(Node *node);
        void draw(Node *node, int depth, const char *edge);
        void traverse_inorder(Node *node, void (*f)(Node *));

        int height(const Node *node) const;
        const Node *find_min(const Node *node) const;
        const Node *find_max(const Node *node) const;

        Node *clone(const Node *) const;
};

#endif /* #ifndef __BST_2_H__ */


Implementation (bst-2.cpp)
---------------------------

#include <iostream>
#include <algorithm>
using namespace std;

#include "bst-2.h"

BST::~BST()
{
    remove_all_nodes(root);
}

void BST::remove_all_nodes(BST::Node *node)
{
    if (node) {
        remove_all_nodes(node->left);
        remove_all_nodes(node->right);
        delete node;
    }
}

void BST::insert(int x)
{
    root = insert(x, root);
}

void BST::remove(int x)
{
    // Node that, unlike insert(), remove() does not return "Node*".
    // Instead, it was written to take "Node*&" as a parameter.
    remove(x, root);
}
```

```
const BST::Node *BST::lookup(int x) const
{
    return lookup(x, root);
}

/*
 * Insert x into the given binary search tree node.
 * If node is not NULL, node is again returned after inserting x.
 * If node is NULL, a new single-node tree is returned.
 */
BST::Node *BST::insert(int x, BST::Node *node)
{
    if (node == NULL) {
        node = new Node(x);
    }
    else if (x < node->data) {
        node->left = insert(x, node->left);
    }
    else if (x > node->data) {
        node->right= insert(x, node->right);
    }
    else {
        // x is in the tree already.
    }

    return node;
}

/*
 * Returns the node containing x.
 * Returns NULL if x is not in the tree rooted at the node.
 */
const BST::Node *BST::lookup(int x, const BST::Node *node) const
{
    if (node == NULL)
        return NULL;
    else if (x == node->data)
        return node;
    else if (x < node->data)
        return lookup(x, node->left);
    else
        return lookup(x, node->right);
}

/*
 * Draw the BST rotated 90-degree counter-clockwise.
 */
void BST::draw()
{
    draw(root, 0, "--");
}

void BST::draw(BST::Node *node, int depth, const char *edge)
{
    if (node) {
        draw(node->right, depth + 1, "/-");
```

```
        for (int i = 0; i < depth; i++) {
            cout << "      ";
        }
        cout << edge << " " << node->get() << endl;

        draw(node->left, depth + 1, "\\-");
    }
}

/*
 * In-order traversal.
 */
void BST::traverse_inorder(void (*f)(BST::Node *))
{
    traverse_inorder(root, f);
}

void BST::traverse_inorder(BST::Node *node, void (*f)(BST::Node *))
{
    if (node) {
        traverse_inorder(node->left, f);
        f(node);
        traverse_inorder(node->right, f);
    }
}

int BST::height() const
{
    return height(root);
}

/*
 * Returns the given node's height, which is defined as the length
 * of a longest path from the node to a leaf. (All leaves have height 0.)
 */
int BST::height(const BST::Node *node) const
{
    if (node == NULL)
        return -1;
    else
        return max(height(node->left), height(node->right)) + 1;
}

/*
 * Find the minimum node.
 */
const BST::Node *BST::find_min() const
{
    return find_min(root);
}

const BST::Node *BST::find_min(const BST::Node *node) const
{
    if (node == NULL)
        return NULL;
    else if (node->left == NULL)
        return node;
    else
```

```
        return find_min(node->left);
}

/*
 * Find the maximum node.
 */
const BST::Node *BST::find_max() const
{
    return find_max(root);
}

const BST::Node *BST::find_max(const BST::Node *node) const
{
    if (node != NULL) {
        while (node->right != NULL)
            node = node->right;
    }
    return node;
}

// Node that, unlike insert(), remove() does not return "Node*".
// Instead, it was written to take "Node*&" as a parameter.
void BST::remove(int x, BST::Node*& t)
{
    if (t == NULL) {
        // x is not found - do nothing.
        return;
    }
    else if (x < t->data) {
        remove(x, t->left);
    }
    else if (t->data < x) {
        remove(x, t->right);
    }
    else if (t->left != NULL && t->right != NULL) {
        // Two-children case:
        // 1. copy over the data from the min node of the right sub-tree
        // 2. recursively remove the node containg the data from the
        //    right sub-tree.
        t->data = find_min(t->right)->data;
        remove(t->data, t->right);
    }
    else {
        // t is either a leaf or has one child.
        Node *oldNode = t;
        t = (t->left != NULL) ? t->left : t->right;
        delete oldNode;
    }
}

BST::BST(const BST& t)
{
    root = clone(t.root);
}

BST& BST::operator=(const BST& rhs)
{
    if (this != &rhs) {
```

```
        remove_all_nodes(root);
        root = clone(rhs.root);
    }
    return *this;
}

BST::Node *BST::clone(const BST::Node *t) const
{
    if (t == NULL) {
        return NULL;
    }

    Node *node = new Node(t->get());
    node->left = clone(t->left);
    node->right = clone(t->right);
    return node;
}
```

```
Test driver (bst-2-test.cpp)
-----------------------------

#include <iostream>
#include <algorithm>
using namespace std;

#include "bst-2.h"

void print_node(BST::Node *node)
{
    cout << node->get() << " ";
}

int main()
{
    srandom(time(NULL));

    // read the number of elements from the user.
    int n;
    cin >> n;

    // Construct an empty tree.
    BST t;

    for (int i = 0; i < n; i++) {
        t.insert(random() % 100);
    }

    cout << "\nThe BST:" << endl;
    t.draw();

    cout << "\nAll nodes in-order:" << endl;
    t.traverse_inorder(print_node);
    cout << endl;

    cout << "\nSome info on the tree:" << endl;
    cout << "height == " << t.height() << endl;;
    cout << "min == " << t.find_min()->get() << endl;
```

```
    cout << "max == " << t.find_max()->get() << endl;

    cout << "\nPerforming some lookups:" << endl;
    for (int i = 0; i < 10; i++) {
        int x = random() % 100;
        const BST::Node *node = t.lookup(x);
        if (node) {
            cout << x << " is in there." << endl;
        } else {
            cout << x << " is NOT in there." << endl;
        }
    }

    // copy-construct a new tree t2 out of t.
    BST t2(t);

    while (1) {
        cout << "\nThe remaining BST:" << endl;
        t.draw();
        cout << "\nType the number to remove (-1 to quit): " << endl;
        int x;
        cin >> x;
        if (cin.eof() || x < 0) {
            break;
        }
        t.remove(x);
    }

    cout << "\nThe original BST:" << endl;
    t2.draw();

    return 0;
}
```