CISS350: Data Structures and Advanced Algorithms
Assignment 10

Objectives:

(a) Implement Binary Search Tree with supporting methods.

Q1.

Implement a binary search tree of integers. (Once you have the tree working, it's easy to convert this to a template class – I'll leave as a DIY exercise.) Values contain in the tree are unique, i.e., there are no duplicate values. As mentioned in class, smaller values are stored in the left subtree and larger values are stored in the right subtree. (This implies that if you want to convert this to a template class, the template type must support the obvious comparison operators.)

The following are some skeleton source files. (Note: As always, skeleton code means the code is incomplete and might contain errors. The purpose is to give some initial guidance and structure of code.)

Note that the nodes have parent pointers and must be set correctly. The only node with `NULL` parent pointer is of course the root. In the case of an empty tree, the `root` is set to `NULL`. Note that the skeleton code is (obviously) incomplete and require either addition or modification.

```
// File: BinarySearchTreeNode.h

#ifndef BINARYSEARCHTREENODE_H
#define BINARYSEARCHTREENODE_H

#include <iostream>

class BinarySearchTreeNode
{
public:
    BinarySearchTreeNode(int key,
                         BinarySearchTreeNode * parent_=NULL,
                         BinarySearchTreeNode * left_=NULL,
                         BinarySearchTreeNode * right_=NULL)
        : key_(key), parent_(parent), left_(left), right_(right)
    {}

    int key() const
    {
        return key_;
    }

    int & key()
    {
        return key_;
    }

    BinarySearchTreeNode * parent() const
    {
        return parent_;
    }

    BinarySearchTreeNode * &  parent()
    {
        return parent_;
    }

    BinarySearchTreeNode * root() const
```

```
    {
        return NULL;
    }

    BinarySearchTreeNode * left() const
    {
        return left_;
    }


    // Returns pointer to the leftmost node of the subtree with this
    // node as root.
    BinarySearchTreeNode * leftmost() const
    {
        return NULL;
    }

    BinarySearchTreeNode * right() const
    {
        return right_;
    }

    BinarySearchTreeNode * & right()
    {
        return right_;
    }

    // Returns pointer to the rightmost node of the subtree with this
    // node as root.
    BinarySearchTreeNode * rightmost() const
    {
        return NULL;
    }

    // Returns true if the node is a leaf node, i.e., the left
    // and right are both NULL.
    bool is_leaf() const
    {
        return true;
    }

    // Returns true if the node is not a leaf node
    bool is_nonleaf() const
    {
        return true;
    }

    // Returns true if the parent is NULL
    bool is_root()
    {
        return true;
    }

private:
    int key_;
    BinarySearchTreeNode * parent_;
    BinarySearchTreeNode * left_;
    BinarySearchTreeNode * right_;
};

std::ostream & operator<<(std::ostream &,
                          const BinarySearchTreeNode &);

void print_inorder(const BinarySearchTreeNode *);
```

```
bool insert(BinarySearchTreeNode * &, int);

#endif
```

```
// File: BinarySearchTreeNode.cpp

#include "BinarySearchTreeNode.h"


std::ostream & operator<<(std::ostream & cout,
                          const BinarySearchTreeNode & node)
{
    cout << "<Node: " << &node
         << ", key:" << node.key()
         << ", parent:" << node.parent()
         << ", left:" << node.left()
         << ", right:" << node.right() << '>';
    return cout;
}


void print_inorder(const BinarySearchTreeNode * node)
{
    if (node == NULL) return;
    print_inorder(node->left());
    std::cout << (*node) << '\n';
    print_inorder(node->right());
}


bool insert(BinarySearchTreeNode * & node, int x)
{
    return true;
}
```

```
// File: BinarySearchTree.h

#ifndef BINARYSEARCHTREE_H
#define BINARYSEARCHTREE_H

#include <vector>
#include "BinarySearchTreeNode.h"

class ValueError{};

class BinarySearchTree
{
public:
    BinarySearchTree()
        : root_(NULL)
    {}

    // Put x[0], x[1], ..., x[size - 1] into the tree
    BinarySearchTree(int x[], int size)
    {}

    // TODO
    BinarySearchTree(const BinarySearchTree & bst)
    {}

    // TODO
    ~BinarySearchTree()
```

```
    {}

    BinarySearchTreeNode * root() const
    {
        return root_;
    }

    // TODO
    // [HINT: Traverse *this and bst.]
    BinarySearchTree & operator=(const BinarySearchTree & bst)
    {
        return (*this);
    }

    BinarySearchTreeNode * get_root() const
    {
        return root_;
    }

    // TODO:
    void clear()
    {
    }

    // Returns true if the tree is empty
    bool is_empty() const
    {
        return true;
    }

    // TODO
    // Returns the height of the tree. Note that the height of an
    // empty tree is -1.
    int height() const
    {
        return 0;
    }

    // TODO:
    // Returns the depth of the node pointer p parameter. This is the number
    // of edges from the root to the node that p points to.
    int depth(BinarySearchTreeNode * p)
    {
        return 0;
    }

    // TODO
    // Returns the number of nodes in this tree.
    int size() const
    {
        return 0;
    }

    // TODO
    // Returns a pointer to the node containing x. If x is not in the tree,
    // NULL is returned.
    BinarySearchTreeNode * find(int x)
    {
        return NULL;
    }

    // TODO
    // Returns the minimum value in the tree. If the tree is empty,
    // a ValueError exception object is thrown.
```

```
int min() const
{
    return 0;
}

// TODO
// Returns the maximum value in the tree. If the tree is empty,
// a ValueError exception object is thrown.
int max() const
{
    return 0;
}

// TODO
// Returns true if *this and t are the same trees, i.e., the
// values and structure of *this and t are the same.
bool operator==(const BinarySearchTree & t)
{
    return true;
}

// TODO
// Inserts x into the tree. If x is already in the tree, a ValueError
// exception object is thrown.
void insert(int x)
{
    bool success = ::insert(root_, x);
    if (!success) throw ValueError();
}

// Remove x from the tree. If x is not in the tree, a ValueError
// exception object is thrown.
void remove(int x)
{}

// TODO
// Returns a vector of integers built by traversing the tree
// using inorder traversal
std::vector< int > inorder() const
{
    std::vector< int > ret;
    return ret;
}

// TODO
// Returns a vector of integers built by traversing the tree
// using preorder traversal
std::vector< int > preorder() const
{
    std::vector< int > ret;
    return ret;
}

// TODO
// Returns a vector of integers built by traversing the tree
// using postorder traversal
std::vector< int > postorder() const
{
    std::vector< int > ret;
    return ret;
}

// TODO
// Returns a vector of integers built by traversing the tree
```

```
// using breadth first traversal
std::vector< int > breadthfirst() const
{
    std::vector< int > ret;
    return ret;
}

// TODO
// Return pointer to k-th ordered node
// [HINT: Use a traversal]
BinarySearchNode * select(int k)
{
    return NULL;
}

// TODO
// Return vector of integers in tree with key values <= m and < M.
std::vector< int > range(int m, int M)
{
    std::vector< int > ret;
    return ret;
}

/*****************************************************************************
// The following iterators are optional. (These are somewhat challenging.)
// This is how the iterator would be used:
//
//   BinarySearchTree::inorder_iterator p = tree.begin();
//   while (p != tree.end())
//   {
//       std::cout << *p << std::endl;
//       p++;
//   }
//
// Note that you should be as memory efficient as possible. So for instance
// Performing a complete traversal, storing a vector of the nodes, and
// iterating through this vector is not allowed.

class inorder_iterator
{
public:
    BinarySearchTreeNode * operator*()
    {}
    void operator++()
    {}
    void operator++(int)
    {}
    void operator--()
    {}
    void operator--(int)
    {}
    // etc.
};

class preorder_iterator
{};

class postorder_iterator
{};

class breadth_first_iterator
{};

*****************************************************************************/
```

```
private:
    BinarySearchTreeNode * root_;
};

void print_inorder(const BinarySearchTree & bst);
void print_preorder(const BinarySearchTree & bst);
void print_postorder(const BinarySearchTree & bst);

#endif
```

```
// File: BinarySearchTree.cpp

#include "BinarySearchTree.h"

void print_inorder(const BinarySearchTree & bst)
{
    print_inorder(bst.root());
}
```

The following tests only some of the functions and methods. You should test all the functions/methods to be implemented.

```
// File: test.cpp

#include <iostream>
#include "BinarySearchTreeNode.h"
#include "BinarySearchTree.h"

int main()
{
    std::cout << "testing BinarySearchTreeNode ...\n";

    BinarySearchTreeNode * n = new BinarySearchTreeNode(5);
    print_inorder(n);

    std::cout << "\n\n";
    insert(n, 3);
    print_inorder(n);

    std::cout << "\n\n";
    insert(n, 0);
    print_inorder(n);

    std::cout << "\n\n";
    insert(n, 1);
    print_inorder(n);

    std::cout << "\n\n";
    insert(n, 4);
    print_inorder(n);

    std::cout << "\n\n";
    insert(n, 8);
    print_inorder(n);

    std::cout << "testing BinarySearchTree ... \n";
    BinarySearchTree bst;
    print_inorder(bst); std::cout << '\n';
```

```
    bst.insert(5); print_inorder(bst); std::cout << '\n';
    bst.insert(0); print_inorder(bst); std::cout << '\n';
    bst.insert(2); print_inorder(bst); std::cout << '\n';
    bst.insert(-2); print_inorder(bst); std::cout << '\n';
    bst.insert(10); print_inorder(bst); std::cout << '\n';
    bst.insert(8); print_inorder(bst); std::cout << '\n';
    bst.insert(9); print_inorder(bst); std::cout << '\n';
    bst.insert(7); print_inorder(bst); std::cout << '\n';
    try
    {
        std::cout << "testing duplicate insert ... ";
        bst.insert(7);
    }
    catch (ValueError & e)
    {
        std::cout << "pass" << std::endl;
    }
    return 0;
}
```

Note that the `insert()` method of the `BinarySearchTree` class uses the `insert()` function in `BinarySearchTreeNode.cpp`. This should be the case for many of the methods in the `BinarySearchTree` class. In software development, we would say that the `BinarySearchTree` class is a wrapper class, i.e., the class wraps up functionality already provided somewhere else.

Note that in the `BinarySearchTree` class, the `insert()` method calls the `insert()` function in the `BinarySearchTreeNode` (the cpp file). In order to achieve this, the function call to the `insert()` function must be referenced as

```
::insert(root_, x)
```

The `::` tells the compiler to look for `insert` in the global scope and not within the class scope of `BinarySearchTree`.

In the case of a failure to insert, the `insert()` function in `BinarySearchTree` returns a `false`; otherwise it returns `true`. On the other hand, the `insert()` method in `BinarySearchTree` throws an exception when an insert is not possible.

Your implementation for `remove()` should be similar, i.e., there should be a `remove()` function in `BinarySearchTreeNode` and a `remove()` method in `BinarySearchTree`. On failure, the `remove()` function `BinarySearchTreeNode` should return `false` whereas the `remove()` method in `BinarySearchTreeNode` should throw an exception.

Q2.

With Q1 done, you can now make your BST objects self-balancing by including balancing after an insert or a remove operation. Make sure check the notes for left and right rotations and when/how they are used after a BST insert and a BST remove.

The name of the class is `AVL` and it should have all the methods in the BST class from Q1. Like the BST case, you only need to write AVL trees of integer values. Note that almost all methods are the same as the methods in BST (Q1).

Make sure you test your AVL insert and remove thoroughly.

(After you're done with Q2, as a personal project, you should convert this into a class template.)

Note. Together with quadtrees, you now have two topics to investigate further and can be used for an undergraduate research project/paper. There are lots of information on quadtree (and their variations and other spatial data structures) and AVL trees (and their variations and other self-balancing trees).