The ADT Binary Search Tree
After Weiss Textbook, Chapter 4

```
#ifndef BINARY_SEARCH_TREE_H
#define BINARY_SEARCH_TREE_H

#include <algorithm>
#include <cassert>     // KV
using namespace std;


template <typename T>
class BinarySearchTree
{
  private:

    struct BinaryNode
    {
        T element;
        BinaryNode *left;
        BinaryNode *right;

        BinaryNode( const T & theElement,
                    BinaryNode *lt, BinaryNode *rt )
          : element{ theElement }, left{ lt },
            right{ rt } { }

        BinaryNode( T && theElement,
                    BinaryNode *lt, BinaryNode *rt )
          : element{ std::move( theElement ) },
            left{ lt }, right{ rt } { }
    };

  public:

    BinarySearchTree( ) : root{ nullptr }
    {
    }

    BinarySearchTree( const BinarySearchTree & rhs )
```

```
   : root{ nullptr }
{
    root = clone( rhs.root );
}


BinarySearchTree( BinarySearchTree && rhs )
    : root{ rhs.root }
{
    rhs.root = nullptr;
}


~BinarySearchTree( )
{
    makeEmpty( );
}

BinarySearchTree & operator=( const
                           BinarySearchTree & rhs )
{
    BinarySearchTree copy = rhs;
    std::swap( *this, copy );
    return *this;
}

BinarySearchTree & operator=( BinarySearchTree &&
                                               rhs )
{
    std::swap( root, rhs.root );
    return *this;
}

const T & findMin( ) const
{
    assert( isEmpty( ) );

    return findMin( root )->element;
}
```

```
const T & findMax( ) const
{
    assert( isEmpty( ) );

    return findMax( root )->element;
}

bool contains( const T & x ) const
{
    return contains( x, root );
}

bool isEmpty( ) const
{
    return root == nullptr;
}

void printTree( ostream & out = cout ) const
{
    if( isEmpty( ) )
        out << "Empty tree" << endl;
    else
        printTree( root, out );
}

void makeEmpty( )
{
    makeEmpty( root );
}


void insert( const T & x )
{
    insert( x, root );
}


void insert( T && x )
{
    insert( std::move( x ), root );
    }

    void remove( const T & x )
    {
        remove( x, root );
    }

  private:

    BinaryNode *root;

    // Internal methods

    void insert( const T & x,
                 BinaryNode * & t )
    {
        if( t == nullptr )
            t = new BinaryNode{ x, nullptr, nullptr };

        else if( x < t->element )
            insert( x, t->left );

        else if( t->element < x )
            insert( x, t->right );

        else
            ;  // Duplicate; do nothing
    }

    void insert( T && x, BinaryNode * & t )
    {
        if( t == nullptr )
            t = new BinaryNode{ std::move( x ),
                                nullptr, nullptr };
        else if( x < t->element )
            insert( std::move( x ), t->left );
        else if( t->element < x )
            insert( std::move( x ), t->right );
        else
            ;  // Duplicate; do nothing
```

```
}


void remove( const T & x,
             BinaryNode * & t )
{
    if( t == nullptr )
        return;   // Item not found; do nothing

    if( x < t->element )
        remove( x, t->left );

    else if( t->element < x )
        remove( x, t->right );

    else if( t->left != nullptr and
             t->right != nullptr ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;

        if (t→left != nullptr)
           t = t->left;
        else
           t = t->right;

        delete oldNode;
    }
}
```

```
BinaryNode * findMin( BinaryNode *t ) const
{
    if( t == nullptr )
        return nullptr;

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

    return findMin( t->left );
}

BinaryNode * findMax( BinaryNode *t ) const
{
    if( t != nullptr )
        while( t->right != nullptr )
            t = t->right;

    return t;
}

bool contains( const T & x,
               BinaryNode *t ) const
{
    if( t == nullptr )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true;    // Match
}


void makeEmpty( BinaryNode * & t )
{
    if( t != nullptr )
    {
        makeEmpty( t->left );
        makeEmpty( t->right );
```

```
            delete t;
        }
        t = nullptr;
    }

    void printTree( BinaryNode *t,
                    ostream & out ) const
    {
        if( t != nullptr )
        {
            printTree( t->left, out );
            out << t->element << endl;
            printTree( t->right, out );
        }
    }


    BinaryNode * clone( BinaryNode *t ) const
    {
        if( t == nullptr )
            return nullptr;
        else
            return new BinaryNode{ t->element,
                clone( t->left ), clone( t->right ) };
    }
};

#endif
```