```
#ifndef BISEARCH_H
#define BISEARCH_H


#include <iostream>
#include <string>

/*
 * Oscar Maldonado
 * LAB07
 * CSE330
 * BinarySearchTree
 *
 * description:
 *
 * In this lab we are going to create a binarysearchTree and we going
to create way to insert and remove and find max/min and keep it a
O(logN)
 *
 */



template <typename Comparable>
class BinarySearchTree
{
public:
    BinarySearchTree(): root{nullptr} { }
    BinarySearchTree(const BinarySearchTree & rhs) { root =
clone(rhs.root); }
    BinarySearchTree(BinarySearchTree && rhs);
    ~BinarySearchTree() { makeEmpty(root); }
    BinarySearchTree & operator=(const BinarySearchTree & rhs);
    BinarySearchTree & operator=(BinarySearchTree && rhs);

    const Comparable & findMin() const;
    const Comparable & findMax() const;
    bool contains(const Comparable & x) const;
    bool isEmpty() const { return root == nullptr; }
    void printTree(std::ostream & out = std::cout) const;

    void makeEmpty() { makeEmpty(root); }
    void insert(const Comparable & x) { insert(x, root); }
    void insert(Comparable && x) { insert(std::move(x), root); }
    void remove(const Comparable & x) { remove(x, root); }
    void inorder() const { inorder(root); }
    void preorder() const {preorder(root);}
    void postorder() const {postorder(root);}
```

```
private:
    struct BinaryNode
    {
        Comparable element;
        BinaryNode * left;
        BinaryNode * right;

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

    BinaryNode * root;

    void insert(const Comparable & x, BinaryNode * & t);
    void insert(Comparable && x, BinaryNode * & t);
    void remove(const Comparable & x, BinaryNode * & t);
    BinaryNode * findMin(BinaryNode * t) const;
    BinaryNode * findMax(BinaryNode * t) const;
    bool contains(const Comparable & x, BinaryNode * t) const;
    void makeEmpty(BinaryNode * & t);
    void printTree(std::ostream & out, BinaryNode * t, std::string
indent, const std::string & tag) const; // added indent and tag
    BinaryNode * clone(BinaryNode * t) const;
    void inorder(BinaryNode * t) const;
    void postorder(BinaryNode* t) const;
    void preorder(BinaryNode* t) const;
};
//-------------------------------------------------------------------
-------------------------------------------------
template <typename Comparable>
BinarySearchTree<Comparable>::BinarySearchTree(BinarySearchTree &&
rhs)
{
    root = rhs.root; // root = move(rhs.root); is not necessary since
root is a pointer (primitive type)
    rhs.root = nullptr;
}
//-------------------------------------------------------------------
-------------------------------------------------
template <typename Comparable>
BinarySearchTree<Comparable> &
BinarySearchTree<Comparable>::operator=(const BinarySearchTree & rhs)
{
    BinarySearchTree copy = rhs; // uses copy constructor
    swap(*this, copy);
```

```
    return *this;
}
//-------------------------------------------------------------------------
-------------------------------------------------
template <typename Comparable>
BinarySearchTree<Comparable> &
BinarySearchTree<Comparable>::operator=(BinarySearchTree && rhs)
{
    swap(root, rhs.root);
    return *this;
}
//-------------------------------------------------------------------------
-------------------------------------------------
template <typename Comparable>
void BinarySearchTree<Comparable>::printTree(std::ostream & out,
BinaryNode * t, std::string indent, const std::string & tag) const
{
    if (t == nullptr)
        return;

    out << indent << tag << t->element << std::endl;
    indent += "    ";
    printTree(out, t->left, indent, "L ");
    printTree(out, t->right, indent, "R ");
}
//-------------------------------------------------------------------------
-------------------------------------------------
template <typename Comparable>
void BinarySearchTree<Comparable>::printTree(std::ostream & out) const
{
    std::cout << "Print Tree\n";
    printTree(out, root, "", "");
}
//-------------------------------------------------------------------------
-------------------------------------------------
template <typename Comparable>
void BinarySearchTree<Comparable>::inorder(BinaryNode * t) const
{
    if (t == nullptr)
        return;

    inorder(t->left);

    std::cout << t->element << " ";
   inorder(t->right);
}
void BinarySearchTree<Comparable>::preorder(BinaryNode * t) const
{
    if (t == nullptr)
        return;
```

```
 std::cout << t->element << " ";
    preorder(t->left);


   preorder(t->right);
}
void BinarySearchTree<Comparable>::postorder(BinaryNode * t) const
{
    if (t == nullptr)
        return;

    postorder(t->left);


   postorder(t->right);
std::cout << t->element << " ";
}

//-------------------------------------------------------------------------
-----------------------------------------------
    template <typename Comparable>
bool BinarySearchTree<Comparable>::contains(const Comparable &x)
const //
{
    return contains(x,root);
}
//-------------------------------------------------------------------------
-----------------------------
template <typename Comparable>
bool BinarySearchTree<Comparable>::contains(const Comparable &x,
BinarySearchTree<Comparable>::BinaryNode *t) const
{
    if(t == nullptr)
    {
        return false;
    }
    else if(x < t->element)
        return contains(x,x->left);
    else if(t->element < x)
        return contains(x,t->right);
    else
        return true;
}
//-------------------------------------------------------------------------
-----------------------------------------------
template <typename Comparable>
void BinarySearchTree<Comparable>::insert(const Comparable&
x,BinaryNode*& t)
{
if(t == nullptr)
```

```
    t = new BinaryNode{x, nullptr, nullptr};
    else if(x < t->element)
        insert (x,x->left);
    else if(t->element < x)
        insert(x,t->right);
    else
        ;

}
//-------------------------------------------------------------------------
-------------------------------------------------
template <typename Comparable>
void BinarySearchTree<Comparable>::insert(Comparable&& 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
    ;
}
//-------------------------------------------------------------------------
-------------------------------------------------
template <typename Comparable>
typename BinarySearchTree<Comparable>::BinaryNode* findMin( typename
BinarySearchTree<Comparable>::BinaryNode* t)
{
    if(t == nullptr)
        return nullptr;
    if(t->left == nullptr)
        return t;
    return  findMin(t->left);
}
//-------------------------------------------------------------------------
-------------------------------------------------
template <typename Comparable>
typename BinarySearchTree<Comparable>::BinaryNode* findMax( typename
BinarySearchTree<Comparable>::BinaryNode* t)
{
    if(t == nullptr)
        return nullptr;
    if(t->right == nullptr)
        return t;
    return  findMax(t->right );
}
//-------------------------------------------------------------------------
-------------------------------------------------
```

```
template <typename obj>
void BinarySearchTree<obj>::remove(const obj &x,
BinarySearchTree<obj>::BinaryNode *&t)
{
    if(t == nullptr)
        return;
    if(x < t->left)
        remove(x,t->left);
    else if(t->element < x)
        remove(x,t->right);
    else if (t->left != nullptr && t->right != nullptr)
    {
        t->element = findMin(t->right)->element;
        remove(t->element,t->right);
    }
    else
    {
        BinaryNode* oldNode = t;
        t = (t->left != nullptr) ? t->left : t->right;
        delete oldNode;
    }
}
//-------------------------------------------------------------------------
-------------------------------------------------
template <typename obj>
void
BinarySearchTree<obj>::makeEmpty(BinarySearchTree<obj>::BinaryNode
*&t)
{
    if(t != nullptr)
    {
        makeEmpty(t->left);
        makeEmpty(t->right);
        delete t;
    }
    t = nullptr;
}
//-------------------------------------------------------------------------
-------------------------------------------------
template <typename obj>
typename BinarySearchTree<obj>::BinaryNode *
BinarySearchTree<obj>::clone(BinarySearchTree<obj>::BinaryNode *t)
const
{
    if(t == nullptr)
        return nullptr;
    else
        return new BinaryNode{t->element,clone(t->left), clone(t-
>right)};
}
```

```
//-------------------------------------------------------------------------
-----------------------------------------------
#endif
```