```
#ifndef SET_H
#define SET_H



#include <iostream>
#include <cassert>

using namespace std;

template <typename Comparable> class Set_iterator;

template <typename Comparable>
class Set
e
public:
    typedef Set_iterator<Comparable> iterator;

    Set(): root{nullptr}, theSize{0} { }
    Set(const Set & rhs) { root = clone(rhs.root); theSize =
rhs.theSize; }
    Set(Set && rhs);
    ~Set() { makeEmpty(root); }
    Set & operator=(const Set & rhs);
    Set & operator=(Set && rhs);

    const Comparable & findMin() const;
    unsigned int count(const Comparable & x) const;
    bool empty() const { return root == nullptr; }
    unsigned int size() const { return theSize; }
    void printTree(ostream & out = cout) const;

    void clear() { makeEmpty(root); theSize = 0; }
    iterator insert(const Comparable & x) { return insert(x, nullptr,
root); }
    iterator insert(Comparable && x) { return insert(move(x), nullptr,
root); }
    void erase(const Comparable & x) { erase(x, root); }
    void erase(iterator itr);

    iterator begin() const; // for in-order traversal
    iterator end() const { return iterator(nullptr); }
    iterator find(const Comparable & x) const { return find(x,
root); }

private:
    struct BinaryNode
    {
        Comparable element;
        BinaryNode * parent;
```

```
        BinaryNode * left;
        BinaryNode * right;

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

    BinaryNode * root;
    unsigned int theSize;

    iterator insert(const Comparable & x, BinaryNode * p, BinaryNode *
& t);
    iterator insert(Comparable && x, BinaryNode * p, BinaryNode * &
t);
    void erase(const Comparable & x, BinaryNode * & t);
    BinaryNode * findMin(BinaryNode * t) const;
    unsigned int count(const Comparable & x, BinaryNode * t) const;
    void makeEmpty(BinaryNode * & t);
    void printTree(ostream & out, BinaryNode * t, string indent, const
string & tag) const; // added indent and tag
    BinaryNode * clone(BinaryNode * t) const;
    iterator find(const Comparable & x, BinaryNode * t) const;

    friend class Set_iterator<Comparable>;
};
//-----------------------------------------------------------------------
---------------------------------------

template <typename Comparable>
typename Set<Comparable>::BinaryNode *
Set<Comparable>::clone(BinaryNode * t) const
{
if (t == nullptr)
        return nullptr;
    else {
        BinaryNode * temp = new BinaryNode{t->element, nullptr,
clone(t->left), clone(t->right)};
        assert(temp);
        if (temp->left != nullptr)
            temp->left->parent = temp;
        if (temp->right != nullptr)
            temp->right->parent = temp;
        return temp;
    }
```

```
}
//-------------------------------------------------------------------------
-----------------------------------

template <typename Comparable>
Set<Comparable>::Set(Set && rhs): root{rhs.root}, theSize{rhs.theSize}
{
    rhs.root = nullptr;
    rhs.theSize = 0;
}
//-------------------------------------------------------------------------
-----------------------------------

template <typename Comparable>
void Set<Comparable>::makeEmpty(BinaryNode * & t)
{
    if (t != nullptr)
    {
        makeEmpty(t->left);
        makeEmpty(t->right);
        delete t;
    }
    t = nullptr;
}
//-------------------------------------------------------------------------
-----------------------------------

template <typename Comparable>
Set<Comparable> & Set<Comparable>::operator=(const Set & rhs)
{
    Set copy = rhs; // uses copy constructor
    swap(*this, copy);
    return *this;
}
//-------------------------------------------------------------------------
-----------------------------------

template <typename Comparable>
Set<Comparable> & Set<Comparable>::operator=(Set && rhs)
{
    swap(root, rhs.root);
    swap(theSize, rhs.theSize);
    return *this;
}
//-------------------------------------------------------------------------
-----------------------------------

template <typename Comparable>
const Comparable & Set<Comparable>::findMin() const
{
```

```
    BinaryNode * t = findMin(root);

    if (t == nullptr)
        return Comparable(); // better raise an exception
 return t->element;
}
//-------------------------------------------------------------------------
---------------------------------------

template <typename Comparable>
unsigned int Set<Comparable>::count(const Comparable & x, BinaryNode *
t) const
{
    if (t == nullptr)
        return 0;
    else if (x < t->element)
        return count(x, t->left);
    else if (x > t->element)
        return count(x, t->right);
    else
        return 1;
}
//-------------------------------------------------------------------------
---------------------------------------

template <typename Comparable>
typename Set<Comparable>::BinaryNode *
Set<Comparable>::findMin(BinaryNode * t) const
{
    if (t == nullptr)
        return nullptr;

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

    return findMin(t->left); // left-slide
}
//-------------------------------------------------------------------------
---------------------------------------

template <typename Comparable>
unsigned int Set<Comparable>::count(const Comparable & x) const
{
    return count(x, root);
}


//-------------------------------------------------------------------------
---------------------------------------
template <typename Comparable>
```

```
void Set<Comparable>::printTree(ostream & out) const
{
    cout << "Print Tree\n";
    printTree(out, root, "", "");
}
//-----------------------------------------------------------------------
---------------------------------------
template <typename Comparable>
void Set<Comparable>::printTree(ostream & out, BinaryNode * t, string
indent, const string & tag) const
{
    if (t == nullptr)
        return;

    out << indent << tag << t->element << endl;
    indent += "    ";
    printTree(out, t->left, indent, "L ");
    printTree(out, t->right, indent, "R ");
}
//-----------------------------------------------------------------------
---------------------------------------
template <typename Comparable>
typename Set<Comparable>::iterator Set<Comparable>::insert(const
Comparable & x, BinaryNode * p, BinaryNode * & t)
{
    if (t == nullptr) {
        t = new BinaryNode{ x, p, nullptr, nullptr };
        assert(t);
        theSize++;
        return t;
    }
    else if (x < t->element)
        return insert(x, t, t->left);
    else if (x > t->element)
        return insert(x, t, t->else
        return t; // duplicate, do nothing
}
//-----------------------------------------------------------------------
---------------------------------------
template <typename Comparable>
void Set<Comparable>::erase(const Comparable & x, BinaryNode * & t)
{
    if (t == nullptr)
        return; // not found, do nothing

    if (x < t->element)
        erase(x, t->left);
    else if (x > t->element)
        erase(x, t->right);
    else if (t->left != nullptr && t->right != nullptr) // two
```

```
children
    {
        t->element = findMin(t->right)->element; // findMin() performs
left-slide from right
        erase(t->element, t->right);
    } else { // one or no child
        BinaryNode *oldNode = t;
        BinaryNode *p = t->parent;
        t = (t->left != nullptr) ? t->left : t->right;
        if (t != nullptr)
            t->parent = p;
        delete oldNode;
        theSize--;
    }
}
//-------------------------------------------------------------------------
---------------------------------------
template <typename Comparable>
void Set<Comparable>::erase(Set<Comparable>::iterator itr) // itr is
undefined on function return
{
    if (itr == end())
        return; // do nothing

    BinaryNode * temp;
    if (itr.current->left != nullptr && itr.current->right !=
nullptr) // two children
    {
        // left-slide from right
        temp = itr.current->right;
        while (temp->left != nullptr)
            temp = temp->left;
        itr.current->element = temp->element;
    } else // one or no child
        temp = itr.current;

    // adjust child pointer
    BinaryNode *p = temp->parent;
);
    else
p->right = (temp->left != nullptr) ? temp->left : temp->right;

    // adjust parent pointer
    if (temp->left != nullptr)
        temp->left->parent = p;
    else if (temp->right != nullptr)
        temp->right->parent = p;

    delete temp;
    theSize--;
```

```
}
//-------------------------------------------------------------------------
----------------------------------------
template <typename Comparable>
typename Set<Comparable>::iterator Set<Comparable>::begin() const
{
    BinaryNode * current = root;
    while (current and current->left)
        current = current->left;
    return iterator(current);
}
//-------------------------------------------------------------------------
----------------------------------------
template <typename Comparable>
class Set_iterator
{
public:
    typedef typename Set<Comparable>::BinaryNode BinaryNode;

    Set_iterator() = default;
    Set_iterator(const Set_iterator & it) = default;
    Set_iterator(Set_iterator && it) = default;
    ~Set_iterator() = default;
    Set_iterator & operator=(const Set_iterator & itr) = default;
    Set_iterator & operator=(Set_iterator && itr) = default;

    Set_iterator(BinaryNode * t): current{t} { }

    bool operator==(Set_iterator itr) const { return current ==
itr.current; }
    bool operator!=(Set_iterator itr) const { return current !=
itr.current; }
    Set_iterator & operator++(); // inorder traversal, pre-increment
    Set_iterator operator++(int); // inorder traversal, post-increment
    Comparable & operator*() { return current->element; }

protected:
    BinaryNode * current;

    friend class Set<Comparable>;
};
//-------------------------------------------------------------------------
----------------------------------------
template <typename Comparable>
typename Set<Comparable>::iterator Set<Comparable>::find(const
Comparable & x, BinaryNode * t) const
{
    if (t == nullptr)
        return iterator{}; // return nullptr; also works, note
constructor for return type
```

```
    else if (x < t->element)
        return find(x, t->left);
    else if (x > t->element)
        return find(x, t->right);
    else
        return iterator{t}; // return t; also works
}
//-------------------------------------------------------------------------
----------------------------------------
template <typename Comparable>
Set_iterator<Comparable> Set_iterator<Comparable>::operator++(int) //
inorder traversal, post-increment
{
    Set_iterator<Comparable> clone(*this);
    ++(*this);
    return clone;
}
//-------------------------------------------------------------------------
----------------------------------------
template <typename Comparable>
Set_iterator<Comparable> & Set_iterator<Comparable>::operator++() //
inorder traversal, pre-increment
{
    if (current->right) { // if right child exists, left slide from
right child
        current = current->right;
while (current->left)
            current = current->left;
    } else {
        BinaryNode * child = current;
        current = current->parent;
        while (current && current->right == child) {
            // coming up from the right child, don't visit the parent
            child = current;
            current = current->parent;
        }
    }
    return *this;
}

#endif
```