45/50

```
Printout for cs320-09-A14-BiTreeL.hpp

// File: BiTreeL/BiTreeL.hpp
// Olivia Lara

#ifndef BITREEL_HPP_
#define BITREEL_HPP_

#include <iostream> // ostream.
using namespace std;

template<class T> class LNode; // Forward declaration.

// ========= BiTreeL =========

template<class T>
class BiTreeL {
    friend class LNode<T>;

private:
    LNode<T> *_root;

    BiTreeL(BiTreeL<T> const &rhs);
    // Copy constructor disabled.

public:
    BiTreeL();
    // Post: This tree is initialized to be empty.

    ~BiTreeL();
    // Post: This tree is deallocated.

public:
    void clear();
    // Post: This tree is cleared to the empty tree.

    bool contains(T const &data) const;
    // Post: true is returned if val is contained in this tree; otherwise, false is
returned.

private:
    LNode<T> *copyRoot(BiTreeL<T> const &rhs);
    // Post: A deep copy of the root of rhs is returned.

public:
    bool equals(BiTreeL<T> const &rhs) const;
    // Post: true is returned if this tree equals tree rhs; otherwise, false is returned.
    // Two trees are equal if they contain the same number of equal elements with the
same shape.

    int height() const;
    // Post: The height of the host tree is returned.

    void inOrder(ostream &os) const;
    // Post: An inorder representation of this tree is sent to os.

    void insertRoot(T const &data);
    // Pre: This tree is empty.
    // Post: This tree has one root node containing data.

    bool isEmpty() const;
    // Post: true is returned if this tree is empty; otherwise, false is returned.

    bool leftIsEmpty() const;
    // Pre: This tree is not empty.
    // Post: true is returned if the left subtree of this tree is empty;
    // otherwise, false is returned.

    T const &max() const;
    // Pre: This tree is not empty.
    // Post: The maximum element of this tree is returned.

    int numLeaves() const;
    // Post: The number of leaves of the host tree is returned.

    int numNodes() const;
    // Post: The number of nodes of the host tree is returned.

    BiTreeL & operator=(BiTreeL<T> const &rhs);
    // Post: A deep copy of rhs is returned with garbage collection.

    void postOrder(ostream &os) const;
    // Post: A postorder representation of this tree is sent to os.

    void preOrder(ostream &os) const;
    // Post: A preorder representation of this tree is sent to os.

    void remLeaves();
    // Post: The leaves are removed from this tree.

    T remRoot();
    // Pre: This tree is not empty.
    // Pre: The root of this tree has at least one empty child.
    // Post: The root node is removed from this tree and its element is returned.

    bool rightIsEmpty() const;
    // Pre: This tree is not empty.
    // Post: true is returned if the right subtree of this tree is empty;
    // otherwise, false is returned.

    T const &root() const;
    // Pre: This tree is not empty.
    // Post: The root element of this tree is returned.

    void setLeft(BiTreeL<T> &subTree);
    // Pre: This tree is not empty.
    // Post: The left child of this tree is subTree.
    // The old left child of this tree is deallocated.
    // subTree is the empty tree (cut setLeft, as opposed to copy setLeft).

    void setRight(BiTreeL<T> &subTree);
    // Pre: This tree is not empty.
    // Post: The right child of this tree is subTree.
    // The old right child of this tree is deallocated.
    // subTree is the empty tree (cut setRight, as opposed to copy setRight).

    void setRoot(T const &data);
    // Pre: This tree is not empty.
    // Post: The root element of this tree is changed to data.

    void toStream(ostream &os) const;
    // Post: A string representation of this tree is sent to os.
};

// ========= LNode =========

template<class T>
class LNode {
    friend class BiTreeL<T>;

private:
    LNode *_left;
    T _data;
    LNode *_right;

private:
    LNode(T data);

private:
    void clear();
    bool contains(T const &data) const;
    LNode<T> *copyRoot();
    // Post: A deep copy of this node is returned.
    bool equals(LNode<T> const *rhs) const;
    int height() const;
    void inOrder(ostream &os) const;
    T const &max() const;
    int numLeaves() const;
    int numNodes() const;
    void postOrder(ostream &os) const;
    void preOrder(ostream &os) const;
    void remLeaves();
    T const &root() const;
    void setLeft(BiTreeL<T> &subTree);
    void setRight(BiTreeL<T> &subTree);
    void setRoot(T const &data);
    void toStream(string prRight, string prRoot, string prLeft, ostream &os) const;
};

// ========= Constructors =========

template<class T>
BiTreeL<T>::BiTreeL() :
_root(nullptr) {
}

template<class T>
LNode<T>::LNode(T data) :
_left(nullptr), _data(data), _right(nullptr) {
}

// ========= Destructors =========

template<class T>
BiTreeL<T>::~BiTreeL() {
    clear();
}

// ========= clear =========

template<class T>
void BiTreeL<T>::clear() {
    _root = nullptr;
    delete _root;
}
```

```
template<class T>
void LNode<T>::clear() {
    if (_left != nullptr) {
        _left = nullptr;
        delete _left;
    }
    if (_right != nullptr) {
        _right = nullptr;
        delete _right;
    }
}
```

```
// ========= clear =========
template<class T>
void BiTreeL<T>::clear() {
    if (_root != nullptr) {
        _root->clear();
        _root = nullptr;
    }
}

template<class T>
void LNode<T>::clear() {
    if (_left != nullptr) {
        _left->clear();
        _left = nullptr;
    }
    if (_right != nullptr) {
        _right->clear();
        _right = nullptr;
    }
    delete this;
}
```

```
Never gets called!
-2
```

```
// ========= contains =========

template<class T>
bool BiTreeL<T>::contains(T const &data) const {
    return _root == nullptr ? false : _root -> contains(data);
}

template<class T>
bool LNode<T>::contains(T const &data) const {
    return _data == data || (_right != nullptr && _right -> contains(data)) || (_left !=
nullptr && _left -> contains(data));
}

// ========= copyRoot =========

template<class T>
LNode<T> *BiTreeL<T>::copyRoot(BiTreeL<T> const &rhs) {
    return rhs.isEmpty() ? nullptr : rhs._root->copyRoot();
}

template<class T>
LNode<T> *LNode<T>::copyRoot() {
    LNode<T> *result = new LNode<T > (_data);
    if (_left != nullptr) {
        result->_left = _left->copyRoot();
    }
    if (_right != nullptr) {
        result->_right = _right->copyRoot();
    }
    return result;
}

// ========= equals =========

template<class T>
bool BiTreeL<T>::equals(BiTreeL<T> const &rhs) const {
    if (_root == nullptr && rhs._root == nullptr) {
        return true;
    } if (_root != nullptr && rhs._root == nullptr) {
        return false;
    } if (_root == nullptr && rhs._root != nullptr) {
        return false;
    }
    return _root -> equals(rhs._root);
}

template<class T>
bool LNode<T>::equals(LNode<T> const *rhs) const {
    if (_data != rhs -> _data) {
        return false;
    } if (_left == nullptr && rhs -> _left == nullptr) {
        if (_right == nullptr && rhs -> _right == nullptr) {
            return true;
        }
    } else if (_left != nullptr && rhs->_left == nullptr ||_left == nullptr && rhs ->
_left != nullptr) {
        return false;
    } else if (_right != nullptr && rhs->_right == nullptr || _right == nullptr && rhs
-> _right != nullptr) {
        return false;
    } else if (_left != nullptr && rhs -> _left != nullptr) {
        return _left -> equals(rhs -> _left);
    } else if (_right != nullptr && rhs -> _right != nullptr) {
        return _right -> equals(rhs -> _right);
    }
}
```

```
// ========= height =========

template<class T>
int BiTreeL<T>::height() const {
    if (_root) {
        return (1 + _root -> height());
    }
    return 0;
}

template<class T>
int LNode<T>::height() const {
    if (_left > _right) {
        return (1 + _left -> height());
    }
    if (_right > _left) {
        return (1 + _right -> height());
    }
    if (_left == nullptr || _right == nullptr) {
        return 0;
    }
}

// ========= inOrder =========

template<class T>
void BiTreeL<T>::inOrder(ostream &os) const {
    if (_root != nullptr) {
        _root -> inOrder(os);
    }
}

template<class T>
void LNode<T>::inOrder(ostream &os) const {
    if (_left != nullptr) {
        _left-> inOrder(os);
    }
    os << _data << " ";
    if (_right != nullptr) {
        _right -> inOrder(os);
    }
}

// ========= insertRoot =========

template<class T>
void BiTreeL<T>::insertRoot(T const &data) {
    if (_root != nullptr) {
        cerr << "insertRoot precondition violated: Cannot insert root into a non empty
tree" << endl;
        throw -1;
    }
    _root = new LNode<T>(data);
}

// ========= isEmpty =========

template<class T>
bool BiTreeL<T>::isEmpty() const {
    return _root == nullptr;
}

// ========= leftIsEmpty =========

template<class T>
```

```
bool BiTreeL<T>::leftIsEmpty() const {
    if (_root == nullptr) {
        cerr << "Precondition violated: Cannot test left subtree of an empty tree." <<
endl;
        throw -1;
    }
    return _root->_left == nullptr;
}

// ========= max =========

template<class T>
T const &BiTreeL<T>::max() const {
    if (_root == nullptr) {
        cerr << "Precondition violated: An empty tree has no maximum." << endl;
        throw -1;
    }
    return _root->max();
}

template<class T>
T const &LNode<T>::max() const {
    T const *dataTemp = &_data; // To avoid restrictions on T const &_data.
    T const *leftMax = (_left == nullptr) ? dataTemp : &_left->max();
    T const *rightMax = (_right == nullptr) ? dataTemp : &_right->max();
    return (*leftMax > *rightMax)
            ? ((*leftMax > *dataTemp) ? *leftMax : *dataTemp)
            : ((*rightMax > *dataTemp) ? *rightMax : *dataTemp);
}

// ========= numLeaves =========

template<class T>
int BiTreeL<T>::numLeaves() const {
    if (_root) {
        return _root -> numLeaves();
    }
    return 0;
}

template<class T>
int LNode<T>::numLeaves() const {
    int result = 0;
    if (_left == nullptr && _right == nullptr) {
        result += 1;
    }
    if (_left) {
        result += _left -> numLeaves();
    }
    if (_right) {
        result += _right -> numLeaves();
    }
    return result;
}

// ========= numNodes =========

template<class T>
int BiTreeL<T>::numNodes() const {
    return _root == nullptr ? 0 : _root->numNodes();
}

template<class T>
int LNode<T>::numNodes() const {
    int result = 1;
    if (_left != nullptr) {
        result += _left->numNodes();
    }
    if (_right != nullptr) {
        result += _right->numNodes();
    }
    return result;
}

// ========= operator= =========

template<class T>
BiTreeL<T> &BiTreeL<T>::operator=(BiTreeL<T> const &rhs) {
    if (this != &rhs) { // In case someone writes myTree = myTree;
        delete _root;
        _root = copyRoot(rhs);
    }
    return *this;
}

// ========= operator== =========

template<class T>
bool operator==(BiTreeL<T> const &lhs, BiTreeL<T> const &rhs) {
    return lhs.equals(rhs);
}

// ========= operator<< =========

template<class T>
ostream & operator<<(ostream &os, BiTreeL<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= postOrder =========

template<class T>
void BiTreeL<T>::postOrder(ostream &os) const {
    if (_root != nullptr) {
        _root -> postOrder(os);
    }
}

template<class T>
void LNode<T>::postOrder(ostream &os) const {
    if (_left != nullptr) {
        _left -> postOrder(os);
    }
    if (_right != nullptr) {
        _right -> postOrder(os);
    }
    os << _data << " ";
}
// ========= preOrder =========

template<class T>
void BiTreeL<T>::preOrder(ostream &os) const {
    if (_root != nullptr) {
        _root->preOrder(os);
    }
}

template<class T>
void LNode<T>::preOrder(ostream &os) const {
    os << _data << "  ";
    if (_left != nullptr) {
        _left->preOrder(os);
    }
    if (_right != nullptr) {
        _right->preOrder(os);
    }
}

// ========= remLeaves =========

template<class T>
void BiTreeL<T>::remLeaves() {
    if (_root != nullptr) {
        if (_root -> _right == nullptr && _root -> _left == nullptr) {
            delete _root;
            _root = nullptr;
        } else {
            _root -> remLeaves();
        }
    }
}

template<class T>
void LNode<T>::remLeaves() {
    if (_right != nullptr) {
        if (_right -> _right == nullptr && _right -> _left == nullptr) {
            delete _right;
            _right = nullptr;
        } else {
            _right -> remLeaves();
        }
    } if (_left != nullptr) {
        if (_left -> _right == nullptr && _left -> _left == nullptr) {
            delete _left;
            _left = nullptr;
        } else {
            _left -> remLeaves();
        }
    }
}

// ========= remRoot =========

template<class T>
T BiTreeL<T>::remRoot() {
    T result = _root -> _data;
    LNode<T> *temp = _root;
    if (_root == nullptr) {
        cerr << "remRoot precondition violated: Cannot remove root from an empty tree."
<< endl;
        throw -1;
    }
    if (_root -> _left != nullptr && _root -> _right != nullptr) {
        cerr << "Cannot remRoot when both children exists" << endl;
        throw -1;
    }
    if (_root -> _left == nullptr && _root -> _right == nullptr) {
        _root = nullptr;
        delete temp;
        return result;
    } else if (_root -> _left != nullptr && _root -> _right == nullptr) {
        _root = _root -> _left;
        temp -> _left = nullptr;
        delete temp;
        return result;
    } else if (_root -> _right != nullptr && _root -> _left == nullptr) {
        _root = _root -> _right;
        temp -> _right = nullptr;
        delete temp;
        return result;
    }
}

// ========= rightIsEmpty =========

template<class T>
bool BiTreeL<T>::rightIsEmpty() const {
    if (_root == nullptr) {
        cerr << "Precondition violated: Cannot test right subtree of an empty tree." <<
endl;
        throw -1;
    }
    return _root->_right == nullptr;
}

// ========= root =========

template<class T>
T const &BiTreeL<T>::root() const {
    if (_root == nullptr) {
        cerr << "root precondition violated: An empty tree has no root." << endl;
        throw -1;
    }
    return _root->root();
}

template<class T>
T const &LNode<T>::root() const {
    return _data;
}

// ========= setLeft =========

template<class T>
void BiTreeL<T>::setLeft(BiTreeL<T> &subTree) {
    if (_root == nullptr) {
        cerr << "Precondition violated: Cannot set left on an empty tree." << endl;
        throw -1;
    }
    _root->setLeft(subTree);
}

template<class T>
void LNode<T>::setLeft(BiTreeL<T> &subTree) {
    if (_left != nullptr) {
        _left->clear();
    }
    _left = subTree._root;
    subTree._root = nullptr;
}

// ========= setRight =========

template<class T>
void BiTreeL<T>::setRight(BiTreeL<T> &subTree) {
    if (_root == nullptr) {
        cerr << "Precondition violated: Cannot set right on an empty tree." << endl;
        throw -1;
    }
    _root->setRight(subTree);
}

template<class T>
```

```
void LNode<T>::setRight(BiTreeL<T> &subTree) {
    if (_right != nullptr) {
        _right->clear();
    }
    _right = subTree._root;
    subTree._root = nullptr;
}

// ========= setRoot =========

template<class T>
void BiTreeL<T>::setRoot(T const &data) {
    if (_root == nullptr) {
        cerr << "Precondition violated: Cannot set root on an empty tree." << endl;
        throw -1;
    }
    _root -> _data = data;
}

template<class T>
void LNode<T>::setRoot(T const &data) {
    _data = data;
}

// ========= toStream =========

template<class T>
void BiTreeL<T>::toStream(ostream &os) const {
    if (_root == nullptr) {
        os << "*";
    } else {
        _root->toStream("", "", "", os);
    }
}

template<class T>
void LNode<T>::toStream(string prRight, string prRoot, string prLeft, ostream &os) const
{
    if (_right == nullptr) {
        os << prRight << "     -*" << endl;
    } else {
        _right->toStream(prRight + "     ", prRight + "     ", prRight + "    |", os);
    }
    os << prRoot;
    os.fill('-');
    os.width(4);
    os.setf(ios::left, ios::adjustfield);
    os << _data << "|" << endl;
    if (_left == nullptr) {
        os << prLeft << "     -*" << endl;
    } else {
        _left->toStream(prLeft + "    |", prLeft + "     ", prLeft + "     ", os);
    }
}

#endif

// new page
```

```
Testing cs320-09 BiTreeL unit-metrics
=====================================

*

The number of nodes is 0

The number of leaves is 0

The height is 0

     -*
70--|
    |               -*
    |          60--|
    |         |     -*
    |     50--|
    |    |     -*
     40--|
         |          -*
         |     30--|
         |    |     -*
          10--|
              |     -*
               20--|
                    -*

The number of nodes is 7

The number of leaves is 3

The height is 4

// new page
```

```
     -*


               -*
          30--|
         |     -*
     10--|
    |    |     -*
    |     20--|
    |          -*
40--|
     -*
```

The trees are not equal.

// new page

```
                                 -3

// ========= equals =========
template<class T>
bool BiTreeL<T>::equals(BiTreeL<T> const &rhs) const {
    return _root == nullptr ?
        rhs.isEmpty() :
        ! rhs.isEmpty() && _root-> equals(rhs._root);
}

template<class T>
bool LNode<T>::equals(LNode<T> const *rhs) const {
    if (_data != rhs->_data) {
        return false;
    }
    return
        (_left != nullptr && rhs->_left != nullptr ?
         _left->equals(rhs->_left) :
         _left == nullptr && rhs->_left == nullptr)
        &&
        (_right != nullptr && rhs->_right != nullptr ?
         _right->equals(rhs->_right) :
         _right == nullptr && rhs->_right == nullptr);
}
```