```

// COS30008, Problem Set 4, Problem 1, 2024

#pragma once

#include <stdexcept>
#include <algorithm>

template<typename T>
struct BinaryTreeNode
{
    using BNode = BinaryTreeNode<T>;
    using BTreeNode = BNode*;

    T key;
    BTreeNode left;
    BTreeNode right;

    static BNode NIL;

    const T& findMax() const
    {
        if ( empty() )
        {
            throw std::domain_error( "Empty tree encountered." );
        }

        return right->empty() ? key : right->findMax();
    }

    const T& findMin() const
    {
        if ( empty() )
        {
            throw std::domain_error( "Empty tree encountered." );
        }

        return left->empty() ? key : left->findMin();
    }

    bool remove( const T& aKey, BTreeNode aParent )
    {
        BTreeNode x = this;
        BTreeNode y = aParent;

        while ( !x->empty() )
        {
            if ( aKey == x->key )
            {
                break;
            }

            y = x;                                      // new parent

            x = aKey < x->key ? x->left : x->right;
        }

        if ( x->empty() )
        {
            return false;                               // delete failed
        }

        if ( !x->left->empty() )
        {
            const T& lKey = x->left->findMax();         // find max to left
            x->key = lKey;
            x->left->remove( lKey, x );
        }
        else
        {
            if ( !x->right->empty() )
            {
                const T& lKey = x->right->findMin();    // find min to right
                x->key = lKey;
                x->right->remove( lKey, x );
            }
            else
            {
                if ( y != &NIL )                        // y can be NIL
                {
                    if ( y->left == x )
                    {
                        y->left = &NIL;
                    }
                    else
                    {
                        y->right = &NIL;
                    }
                }

                delete x;                               // free deleted node
            }
        }

        return true;
```

```
    }

    // PS4 starts here

    // default constructor
    BinaryTreeNode() :
        key(T()),
        left(&NIL),
        right(&NIL)
    { }

    // constructor with key
    BinaryTreeNode(const T& aKey) :
        key(aKey),
        left(&NIL),
        right(&NIL)
    { }

    // move constructor
    BinaryTreeNode( T&& aKey ) :
        key(std::move(aKey)),
        left(&NIL),
        right(&NIL)
    { }

    // destructor
    ~BinaryTreeNode()
    {
        // delete left and right subtrees
        // if they are not NIL
        if (!left->empty())
        {
            delete left;
        }

        if (!right->empty())
        {
            delete right;
        }
    }

    // Is this node NIL (sentinel)?
    bool empty() const
    {
        return this == &NIL;
    }

    // Is this node a leaf?
    bool leaf() const
    {
        return left->empty() && right->empty();
    }

    // Height of the tree
    size_t height() const
    {
        // If first call is on NIL, throw domain error
        if (empty())
        {
            throw std::domain_error("Empty tree encountered!");
        }

        // Leaf node has height 0
        if (leaf())
        {
            return 0;
        }

        // calculate height of left and right subtrees
        // ignoring NIL nodes (not error)
        size_t lLeftHeight = left->empty() ? 0 : left->height();
        size_t lRightHeight = right->empty() ? 0 : right->height();

        // return 1 + max subtree height
        return 1 + std::max(lLeftHeight, lRightHeight);
    }

    bool insert(const T& aKey)
    {
        // If trying to insert a key into NIL
        // or duplicate key, return false
        if (empty() || aKey == key)
        {
            return false;
        }

        if (aKey < key) // insert left
        {
            if (left->empty())
            {
                // insert new node as left child
                left = new BNode(aKey);
                return true;
            }
            else
            {
                // recursively insert into left subtree
                return left->insert(aKey);
            }
        }
        else // insert right
        {
            if (right->empty())
            {
                // insert new node as right child
                right = new BNode(aKey);
                return true;
            }
            else
            {
                // recursively insert into right subtree
                return right->insert(aKey);
            }
        }
    }
};

template<typename T>
BinaryTreeNode<T> BinaryTreeNode<T>::NIL;

```