

```

// COS30008, Final Exam, 2021

#pragma once

#include <stdexcept>

template<typename T>
class TTreePostfixIterator;

template<typename T>
class TTree
{
private:
    T fKey;
    TTree<T>* fLeft;
    TTree<T>* fMiddle;
    TTree<T>* fRight;

    TTree() : fKey(T()) // use default constructor to initialize fKey
    {
        fLeft = &NIL; // loop-back: The sub-trees of a TTree object with
        fMiddle = &NIL; // no children point to NIL.
        fRight = &NIL;
    }

    void addSubTree(TTree<T>** aBranch, const TTree<T>& aTTree)
    {
        if (!(*aBranch)->empty())
        {
            delete* aBranch;
        }

        *aBranch = const_cast<TTree<T>*>(&aTTree);
    }

public:
    using Iterator = TTreePostfixIterator<T>;

    static TTree<T> NIL; // sentinel

    // getters for subtrees
    const TTree<T>& getLeft() const { return *fLeft; }

```

```

const TTree<T>& getMiddle() const { return *fMiddle; }
const TTree<T>& getRight() const { return *fRight; }

// add a subtree
void addLeft(const TTree<T>& aTTree) { addSubTree(&fLeft, aTTree); }
void addMiddle(const TTree<T>& aTTree) { addSubTree(&fMiddle, aTTree); }
void addRight(const TTree<T>& aTTree) { addSubTree(&fRight, aTTree); }

// remove a subtree, may through a domain error
const TTree<T>& removeLeft() { return removeSubTree(&fLeft); }
const TTree<T>& removeMiddle() { return removeSubTree(&fMiddle); }
const TTree<T>& removeRight() { return removeSubTree(&fRight); }

// Problem 1: TTree Basic Infrastructure

```

private:

```

// remove a subtree, may through a domain error
const TTree<T>& removeSubTree(TTree<T>** aBranch) {
    if (!(*aBranch)->empty())
    {
        *aBranch = &NIL;
    }
    return *this;
}

```

public:

```

// TTree 1-value constructor
TTree(const T& aKey) :
    fKey(aKey)
{
    fLeft = &NIL;      // loop-back: The sub-trees of a TTree object with
    fMiddle = &NIL;    // no children point to NIL.
    fRight = &NIL;
}

// destructor (free sub-trees, must not free empty trees)
~TTree() {
    if (!empty()) {
        removeLeft();
        removeMiddle();
        removeRight();
    }
}

```

```

// return key value, may throw domain_error if empty
const T& operator*() const {
    if (empty())
        throw std::domain_error("Empty TTree encountered.");
    return fKey;
}

// returns true if this TTree is empty
bool empty() const {
    return this == &NIL;
}

// returns true if this TTree is a leaf
bool leaf() const {
    return fLeft->empty() && fMiddle->empty() && fRight->empty();
}

// Problem 2: TTree Copy Semantics

// copy constructor, must not copy empty TTree
TTree(const TTree<T>& aOtherTTree)
{
    if (aOtherTTree.empty()) {
        throw std::domain_error("Copying NIL");
    }
    fLeft = new TTree<T>();
    fRight = new TTree<T>();
    fMiddle = new TTree<T>();
    if (!aOtherTTree.empty()) {
        fKey = aOtherTTree.fKey;
        fLeft = aOtherTTree.fLeft->clone();
        fRight = aOtherTTree.fRight->clone();
        fMiddle = aOtherTTree.fMiddle->clone();
    }
}

// assignment operator, must not copy empty TTree
TTree<T>& operator=(const TTree<T>& aOtherTTree) {
    if (aOtherTTree.empty()) {
        throw std::domain_error("Copying NIL");
    }
    if (**this != *aOtherTTree) {
        //delete

```

```

        delete this;
        //copy
        fKey = std::move(aOtherTTree.fKey);
        fLeft = std::move(aOtherTTree.fLeft);
        fMiddle = std::move(aOtherTTree.fMiddle);
        fRight = std::move(aOtherTTree.fRight);
    }
    return *this;
}

// clone TTree, must not copy empty trees
TTree<T>* clone() const {
    if (!empty()) {
        return new TTree(*this);
    }
}

// Problem 3: TTree Move Semantics

// TTree r-value constructor
TTree(T&& aKey) : fKey(std::move(aKey)), fLeft(&NIL), fMiddle(&NIL),
fRight(&NIL) {}

// move constructor, must not copy empty TTree
TTree(TTree<T>&& aOtherTTree) : fKey(aOtherTTree.fKey),
fLeft(aOtherTTree.fLeft), fRight(aOtherTTree.fRight),
fMiddle(aOtherTTree.fMiddle) {
    if (aOtherTTree.empty()) {
        throw std::domain_error("Moving NIL");
    }
    aOtherTTree.removeLeft();
    aOtherTTree.removeRight();
    aOtherTTree.removeMiddle();
}

//move assignment operator, must not copy empty TTree
TTree<T>& operator=(TTree<T>&& aOtherTTree) {
    if (!aOtherTTree.empty()) {
        if (this != &aOtherTTree) {
            //delete
            if (fLeft != &NIL) {
                delete fLeft;
            }
            if (fMiddle != &NIL) {
                delete fMiddle;
            }
        }
    }
}

```

```

    }
    if (fRight != &NIL) {
        delete fRight;
    }

    fLeft = std::move(aOtherTTree.fLeft);
    fMiddle = std::move(aOtherTTree.fMiddle);
    fRight = std::move(aOtherTTree.fRight);
    //copy
    &aOtherTTree.removeLeft();
    &aOtherTTree.removeMiddle();
    &aOtherTTree.removeRight();
}
return *this;
}
else {
    throw std::domain_error("Moving NIL");
}
}

// Problem 4: TTree Postfix Iterator

// return TTree iterator positioned at start
Iterator begin() const;

// return TTree iterator positioned at end
Iterator end() const;
};

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

```