#ifndef BINARY_TREE_H
#define BINARY_TREE_H

#include <vector>
#include "VariousExceptions.h"

template <typename Object>
template <typename Object>
class BinaryTree {
public:
    /* Default Constructor. * Creates a tree with single node (which contains default element) */
    BinaryTree() : sz(1), rt(new Node) { }
    
    /* Copy Constructor */
    BinaryTree(const BinaryTree& orig) :
        sz(orig.sz), rt(cloneRecurse(orig,orig.rt)) { }
    
    /* Overloaded Assignment Operator */
    BinaryTree& operator=(const BinaryTree& orig) {
        if (this != &orig) {
            clearRecurse(rt);
            rt = cloneRecurse(orig,orig.rt);
        }
        return *this;
    }
    
    /* Destructor */
    ~BinaryTree() {
        clearRecurse(rt);
    }

    /* Constructor */
    BinaryTree(const Object& e = Object(), Node* p = NULL, Node* l = NULL, Node* r = NULL) :
        element(e), parent(p), left(l), right(r) { }

    /* Struct Node:  this represents a single node of the tree */
    struct Node {
        Object element; // element
        Node* parent; // parent node
        Node* left; // left child
        Node* right; // left child
        Node(const Object& e = Object(), Node* p = NULL, Node* l = NULL, Node* r = NULL) :
            element(e), parent(p), left(l), right(r) { }
    }
    
    typedef Node* NodePtr; // pointer to a Node

    /* Class Position:  represents a position in a BinaryTree */
    class Position {
public:
        /* Constructor */
        Position(NodePtr n = NULL) : node(n) { }

        /* Returns the element at the Position */
        Object& element() const throw(InvalidPositionException) {
            if (node == NULL) throw InvalidPositionException("Null position");
            return node->element;
        }

        /* Determines whether the Position is a 'null' position */
        bool isNull() const { return node == NULL; }

        /* Overload the equality operator */
        bool operator==(const Position& other) { return (this->node == other.node); }

        /* Overload the equality operator */
        bool operator!=(const Position& other) { return ! operator==(other); }
    }

private:
    /* Class Node:  this represents a single node of the tree */
    struct Node {
        Object element; // node in the BinaryTree
        Node* parent; // parent node
        Node* left; // left child
        Node* right; // left child
        Node(const Object& e = Object(), Node* p = NULL, Node* l = NULL, Node* r = NULL) :
            element(e), parent(p), left(l), right(r) { }

        typedef Node* NodePtr; // pointer to a Node

        /* Class Position:  represents a position in a BinaryTree */
        class Position {
public:
            /* Constructor */
            Position(NodePtr n = NULL) : node(n) { }

            /* Returns the element at the Position */
            Object& element() const throw(InvalidPositionException) {
                if (node == NULL) throw InvalidPositionException("Null position");
                return node->element;
            }

            /* Determines whether the Position is a 'null' position */
            bool isNull() const { return node == NULL; }

            /* Overload the equality operator */
            bool operator==(const Position& other) { return (this->node == other.node); }

            /* Overload the equality operator */
            bool operator!=(const Position& other) { return ! operator==(other); }
        }

private:
    friend class BinaryTree<Object>; // allow access to private member

    NodePtr validate(const BinaryTree<Object>* tree) const throw(InvalidPositionException) {
        if (node == NULL)
            throw InvalidPositionException("Cannot use a NULL position");

        if (node->tree->rt && node->parent == NULL)
            throw InvalidPositionException("Position appears to involve removed element.");

        else return node;
    }

    ~BinaryTree(); // allow access to private member

    /*********** end of class Position *****************************/
/* Iterator Classes */

template <typename T>
class Iterator {
public:
    /* Are there more items left in iteration? */
    bool hasNext() {
        return (index < items.size());
    }
    /* Returns the next available item in the iteration */
    T next() {
        return items[index++];
    }

    friend class BinaryTree;
    Iterator() : index(0) { }
    std::vector<T> items; // vector of items
    unsigned int index; // current index
};

typedef Iterator<Position> PositionalIterator;
typedef Iterator<Object> ObjectIterator;

/******************** end of iterator classes **********************/

/**************************** query methods ************************/

/* Returns size of tree */
int size() const {
    return sz;
}

/* does position correspond to internal node? */
bool isInternal(const Position& p) const {
    throw(InvalidPositionException) {
        NodePtr v = p.validate(this);
        return v->left != NULL;
    }
}

/* does position correspond to external node? */
bool isExternal(const Position& p) const {
    throw(InvalidPositionException) {
        NodePtr v = p.validate(this);
        return v->left == NULL;
    }
}

/* is this the root position? */
bool isRoot(const Position& p) const {
    throw(InvalidPositionException) {
        NodePtr v = p.validate(this);
        return v == rt;
    }
}

/**************************** accessor methods ************************/

/* return position of root of tree */
Position root() const {
    return Position(rt);
}

/* return position of parent of given position */
Position parent(const Position& p) const {
    throw(BoundaryViolationException, InvalidPositionException) {
        NodePtr v = p.validate(this);
        if (v==rt) throw BoundaryViolationException("Cannot traverse parent of root");
        return Position(v->parent);
    }
}

/* return position of left child of given position */
Position leftChild(const Position& p) const {
    throw(BoundaryViolationException, InvalidPositionException) {
        NodePtr v = p.validate(this);
        if (v->left==NULL) throw BoundaryViolationException("Cannot traverse child of external position");
        return Position(v->left);
    }
}

/* return position of right child of given position */
Position rightChild(const Position& p) const {
    throw(BoundaryViolationException, InvalidPositionException) {
        NodePtr v = p.validate(this);
        if (v->right==NULL) throw BoundaryViolationException("Cannot traverse child of external position");
        return Position(v->right);
    }
/* return position of sibling of given position */
Position sibling(const Position& p) const
{
    throw(BoundaryViolationException, InvalidPositionException) {
        NodePtr v = p.validate(this);
        if (v==rt) throw BoundaryViolationException("Cannot traverse sibling of root");
        NodePtr parent = v->parent;
        NodePtr lc = parent->left;
        NodePtr s = (v==lc ? parent->right : lc);
        return Position(s);
    }
}

/* return iterator of all children of given position */
PositionalIterator children(const Position& p) const
{
    NodePtr v = p.validate(this);
    PositionalIterator PI;
    if (v->left!=NULL) {
        PI.items.push_back(Position(v->left));
        PI.items.push_back(Position(v->right));
    }
    return PI;
}

/* return iterator of all positions of the tree */
PositionalIterator positions() const {
    PositionalIterator PI;
    recurseAdd(PI,rt);
    return PI;
}

/* return iterator of all elements stored in the tree */
ObjectIterator elements() const {
    ObjectIterator OI;
    recurseAdd(OI,rt);
    return OI;
}

/****************************** update methods ****************************/

/* replace the element at the given position */
void replaceElement(const Position& p, const Object& element) throw(InvalidPositionException) {
    NodePtr v = p.validate(this);
    v->element = element;
}

/* swap the elements stored at the given positions */
void swapElements(const Position& p, const Position& q) throw(InvalidPositionException) {
    NodePtr v = p.validate(this);
    NodePtr w = q.validate(this);
    Object e = v->element;
    v->element = w->element;
    w->element = e;
}

/* Converts external position into an internal node with two newly created external children (each of which have default element) */
void expandExternal(const Position& p) throw(InvalidPositionException,BoundaryViolationException) {
    NodePtr v = p.validate(this);
    if (v->left==NULL) {
        v->left = new Node(Object(),v,NULL,NULL);
        v->right = new Node(Object(),v,NULL,NULL);
        sz+=2;
    } else {
        throw BoundaryViolationException("Cannot expand internal node");
    }
}
/* Replaces the external position p with a subtree which mirrors the
* contents of a second tree T2. Existing positions of the second
* tree remain valid in the result.
* Note well: the external node as well as the second tree itself are
* destroyed as a side effect.
*/

void replaceExternalWithSubtree(const Position& p, BinaryTree& T2)

throw(InvalidPositionException,BoundaryViolationException) {
    NodePtr v = p.validate(this);
    if (v->left==NULL) {
        sz+=T2.sz-1;
        if (v==rt) {
            rt = T2.rt;
        } else {
            NodePtr parent = v->parent;
            T2.rt->parent = parent;
            if (v==parent->left) {
                parent->left = T2.rt;
            } else {
                parent->right = T2.rt;
            }
        }
    } else {
        delete v;
    }

    // convert T2 back to a default tree (in a way so that its
    // original nodes are not destroyed, as they are now part of
    // this tree)
    T2.sz = 1;
    T2.rt = new Node;
    } else {
        throw BoundaryViolationException("Cannot replace internal node");
    }
}

/* Takes an external position w of tree, deletes w and the parent of
* w from the tree, promoting the sibling of w into the parent’s
* place (see Figure 6.13)
*/

Position removeAboveExternal(const Position& w)

throw(InvalidPositionException,BoundaryViolationException) {
    NodePtr v = w.validate(this);
    if (v==rt) {
        throw BoundaryViolationException("Cannot use removeAboveExternal on root");
    } else {
        throw BoundaryViolationException("Cannot use removeAboveExternal on internal node");
    }

    NodePtr parent = v->parent;
    NodePtr s = sibling(w).node;
    if (isRoot(parent)) {
        rt = s;
    } else {
        NodePtr grand = parent->parent;
        if (parent==grand->left) {
            grand->left = s;
        } else {
            grand->right = s;
            s->parent = grand;
        }
        sz-=2;
        delete parent;
        delete v;
        return Position(s);
    }

private:

int sz; // number of items
NodePtr rt; // root of the tree

/* Utilities used for tree iterators */

void recurseAdd(PositionalIterator &pi, const NodePtr v) const {
    if (v->left!=NULL) recurseAdd(pi,v->left);
    pi.items.push_back(Position(v));
    if (v->right!=NULL) recurseAdd(pi,v->right);
}

void recurseAdd(ObjectIterator &oi, const NodePtr v) const {
    if (v->left!=NULL) recurseAdd(oi,v->left);
    oi.items.push_back(v->element);
    if (v->right!=NULL) recurseAdd(oi,v->right);
}
void clearRecurse(const NodePtr v) {
    if (v!=NULL) {
        if (v->left!=NULL) {
            clearRecurse(v->left);
            clearRecurse(v->right);
        }
        delete v;
    }

    NodePtr cloneRecurse(const BinaryTree& orig, const NodePtr v) {
        NodePtr n = new Node(v->element);
        if (v->left!=NULL) {
            n->left = cloneRecurse(orig,v->left);
            n->left->parent = n;
            n->right = cloneRecurse(orig,v->right);
            n->right->parent = n;
        }
        return n;
    }

    NodePtr cloneRecurse(const BinaryTree& orig, const NodePtr v) {
        NodePtr n = new Node(v->element);
        if (v->left!=NULL) {
            n->left = cloneRecurse(orig,v->left);
            n->left->parent = n;
            n->right = cloneRecurse(orig,v->right);
            n->right->parent = n;
        }
        return n;
    }

    #endif