// FILE: bintree.h
// PROVIDES: A classical style class for a node in a binary tree of chars
// functions for manipulating binary trees implemented with linked nodes.
// each node.

// CONSTRUCTOR for the bnode class:
// bnode(
//     char ch,
//     bnode * ipleft = NULL,
//     bnode * ipright = NULL
// )
// Postcondition: The new node has its data equal to ch,
// and its child pointers equal to ipleft ipright.

#ifndef BINTREE_H
#define BINTREE_H
#include <cstdlib>  // Provides NULL and size_t
#include <cassert>
#include <iostream>
class bnode
{
public:
    //data members  Representation invariants:
    char data;    //==data in this node
    bnode *pleft; //==NULL or addr. of root node of left subtree, and
    //all bnodes in the left subtree satisfy this invariant.
    bnode *pright;//==NULL or addr. of root node of right subtree, and
    //all bnodes in the right subtree satisfy this invariant.

    //CONSTRUCTOR, node inline implementation.
    bnode(
        char ch,
        bnode * ipleft = NULL,
        bnode * ipright = NULL
    )

    bool is_leaf( )
    // Postcondition: The return value is true if the node is a leaf;
    // otherwise the return value is false.

    void inorderPrint() const
    // Precondition: node properly constructed.
    // Postcondition: The tree rooted at *this is printed in inorder
    // order, with proper indentation; that is, the char in
    // each node with depth d is printed indented by d spaces.

    bool find( const char& ch )
    // PRECONDITION: *this IS the root of a BINARY SEACH TREE
    // see textbook or notes for that definition!
    // POSTCONDITON: returns true if ch is in the search tree,
    // false if not.

    void insert(const char& ch )
    // PRECONDITION: (1) *this IS the root of a BINARY SEACH TREE
    // see textbook or notes for that definition!
    // (2) ch is NOT ALREADY IN this tree
    // POSTCONDITION: ch is inserted properly, so *this
    // is STILL (the root of) a BINARY SEARCH TREE

};
#endif
data = ch;
pleft = ipleft;
pright = ipright;

//CONST MEMBER FUNCTIONS
bool is_leaf() const
{
    return (pleft == NULL) && (pright == NULL);
}

void inorderPrint () const { doinorderPrint( 0 ); }
void preorderPrint () const { dopreorderPrint( 0 ); }
void postorderPrint () const { dopostorderPrint( 0 ); }

std::size_t tree_size()
{
    std::size_t ret = 1; //Count me first!
    if(pleft) ret += pleft->tree_size();
    if(pright) ret += pright->tree_size();
    return ret;
}

bool find (const char& ch) const
{
    if( ch == data ) return true;
    if( ch < data )
    {
        if( pleft ) return pleft->find(ch);
        else return false;
    }
    else //INVARIANT: data < ch.
    {
        if( pright ) return pright->find(ch);
        else return false;
    }
}

void insert (const char& ch)
{
    assert( find(ch) == false );
    bnode *pParent = findPlacesParent( ch );
    bnode *pnew = new bnode( ch );
    if( ch < pParent->data ) pParent->pleft = pnew; 
    else pParent->pright = pnew;
}

private:
//PRIVATE MEMBER FUNCTIONS (helpers)
void doinorderPrint ( std::size_t depth ) const
//PRECONDITION: *this is a legally build bnode, depth>=0
//POSTCONDITION: The tree rooted at *this is printed
//in inorder order, properly indented, with the
//WHOLE THING also indented depth spaces.
{
    if(pleft) pleft->doinorderPrint( depth + 1 );
    for( std::size_t i=0; i<depth; i++ ) std::cout << ' '; 
    std::cout << data << std::endl;
    if(pright) pright->doinorderPrint( depth + 1 );
}

void dopreorderPrint ( std::size_t depth ) const;
void dopostorderPrint ( std::size_t depth ) const;

bnode * findPlacesParent( const char& ch );
//implemented in bintree.cxx
//PRECONDITION: (1) *this is the root of a BINARY SEARCH TREE
// (2) ch is NOT IN this tree.
//POSTCONDITION: The address of the node under which ch should
// be stored, under binary search tree rules, is returned.
); #endif