Pair Programming 8 Turn In

Name: \_\_\_\_\_\_Braden Bell\_\_\_\_\_ Username: \_\_c1020a01\_\_\_\_\_\_\_\_

Partner name: \_\_\_\_\_Thomas Snyder\_\_\_\_\_\_\_\_\_ Partner username: \_\_\_c1020a10\_\_\_\_\_\_\_\_

\_X\_ I certify that my partner worked with me on this assignment.

SCORE: \_\_\_\_\_\_\_\_\_\_\_\_ (to be filled in by instructor)

8a (7 points)

**[PASTE THE SCREEN SHOT OF THE TESTS WITH A WHITE BACKGROUND FOR PAIR PROGRAMMIG 8A HERE]**

Text

Description automatically generated with low confidence

**[PASTE CODE FOR PAIR PROGRAMMING 8A HERE – NOT A SCREEN SHOT. PASTE node.h, node.cpp. linkedlist.h, AND linkedlist.cpp. HERE. MAKE SURE THE CODE IS SINGLE SPACED AND USE A COURIER NEW FONT]**

**Node.h:**#ifndef NODE\_H

#define NODE\_H

#include "book.h"

class Node {

private:

Book book;

Node\* nextPtr;

Node\* prevPtr;

public:

Node();

Node( const Book& );

Node( const Node& );

virtual ~Node() {}

void setBook( const Book& );

Book getBook() const { return book; }

// LinkedList is a friend

friend class LinkedList;

};

#endif

**Node.cpp:**

#include <cstdlib>

#include "node.h"

#include "book.h"

/\* default constructor

\*/

Node::Node() {

nextPtr = (Node\*)&prevPtr;

}

/\* 1-arg copy constructor doesn't copy the pointer

\*/

Node::Node( const Node& n ) : nextPtr( NULL ) {

setBook( n.book );

}

/\* 1-arg constructor

\* Parameter the book data

\*/

Node::Node( const Book& b ) : nextPtr( NULL ), prevPtr(NULL) {

setBook(b);

}

/\* setBook sets the Book data values

\* Parameter the book data

\* Returns: nothing

\*/

void Node::setBook( const Book& b ) {

book.setIsbn( b.isbn );

book.setAuthor( b.author );

book.setTitle( b.title );

book.setPublisher( b.publisher );

book.setYear( b.year );

book.setEdition( b.edition );

}

**Linkedlist.h:**

#ifndef LINKEDLIST\_H

#define LINKEDLIST\_H

#include "book.h"

#include "node.h"

class LinkedList {

public:

LinkedList();

LinkedList( const LinkedList& );

void insertNode( Node\* ); // at head

void traverseQueue() const;

void traverseStack() const;

Node\* findNode( const Book& ) const;

void deleteNode( Node\* );

Node\* dequeueNode(); // from tail

Node\* popNode(); // from head

virtual ~LinkedList() { deleteList(); }

bool isEmpty() const { return headPtr == NULL; }

LinkedList& operator=( const LinkedList& );

void deleteList();

friend class Node;

private:

Node\* headPtr;

Node\* tailPtr;

int count;

};

#endif

**Linkedlist.cpp:**

/\* File: linkedList.cpp

\* Author: Cindy

\* Description: implements a linked list

\*/

#include <cstdlib>

#include "book.h"

#include "node.h"

#include "linkedlist.h"

LinkedList::LinkedList() {

headPtr = NULL;

tailPtr = NULL;

count = 0;

}

//Write a deep copy constructor

LinkedList::LinkedList(LinkedList const& l) {

// thisPtr goes through this' list, curPtr goes thru l's

Node\* thisPtr = NULL, \*curPtr = l.headPtr;

headPtr = NULL;

if ( l.headPtr != NULL ) { // if there's list to copy

curPtr = l.headPtr;

headPtr = new Node( \*(l.headPtr) );

thisPtr = headPtr;

curPtr = curPtr->nextPtr;

while ( curPtr != NULL ) {

thisPtr->nextPtr = new Node( \*curPtr );

thisPtr->nextPtr->prevPtr = thisPtr;

thisPtr = thisPtr->nextPtr;

curPtr = curPtr->nextPtr;

}

}

tailPtr = thisPtr;

count = l.count;

//return \*this;

}

/\* insertNode inserts a node at the beginning of the list

\* Parameter a pointer to the node to insert

\* Precondition: parameter nodePtr has been allocated

\* Returns: nothing

\*/

void LinkedList::insertNode( Node\* nodePtr ) {

if (headPtr != NULL) {

nodePtr->nextPtr = headPtr;

headPtr->prevPtr = nodePtr;

headPtr = nodePtr;

}

else {

headPtr = tailPtr = nodePtr;

}

count++;

}

/\* traverseStack traverses entire list from head to tail and prints every node's book

\*/

void LinkedList::traverseStack() const {

Node\* curPtr = headPtr;

while (curPtr != NULL ) {

cout << curPtr->book << endl;

curPtr = curPtr->nextPtr;

}

}

/\* traverseQueue traverses entire list from tail to head and prints every node's book

\*/

void LinkedList::traverseQueue() const {

Node\* curPtr = tailPtr;

while (curPtr != NULL ) {

cout << curPtr->book << endl;

curPtr = curPtr->prevPtr;

}

}

/\* finds a node in the list that has data in it and return

\* pointer or NULL if not found

\* Parameters:

\* b book to be found

\* Returns: pointer to node where book is found or NULL

\*/

Node\* LinkedList::findNode( const Book& b ) const {

Node\* curPtr = headPtr;

while( curPtr != NULL ) {

if ( curPtr->book == b )// USES OVERLOADED == BOOK OPERATOR!

return curPtr;

curPtr = curPtr->nextPtr;

}

return NULL;

}

/\* deleteNode deletes node from list pointed to by parameter and deallocates it

\* Parameter pointer to node to delete

\* Precondition nodePtr points to a node in the list

\* Postcondition list altered by node removal

\*/

void LinkedList::deleteNode( Node\* nodePtr ) {

if ( headPtr != NULL ) { // do nothing if list empty

// node at beginning of list?

if ( headPtr == nodePtr ) {

headPtr = headPtr->prevPtr;

headPtr->prevPtr = NULL;

delete nodePtr;

count--;

if ( count == 0 ) tailPtr = NULL; // removed last node?

}

// node at end of list and list has more than one node

else if ( tailPtr == nodePtr ) {

tailPtr = tailPtr->prevPtr;

tailPtr->nextPtr = NULL;

delete nodePtr;

count--;

}

else { // node in middle of list

nodePtr->prevPtr->nextPtr = nodePtr->nextPtr;

nodePtr->nextPtr->prevPtr = nodePtr->prevPtr;

count--;

delete nodePtr;

}

}

}

/\* popNode deletes node at front of list

\* Postcondition list altered by node removal

\*/

Node\* LinkedList::popNode() {

Node\* oldPtr = headPtr;

if ( oldPtr != NULL ) {

headPtr = headPtr->nextPtr;

if ( headPtr != NULL ) headPtr->prevPtr = NULL;

count--;

return oldPtr;

}

return NULL;

}

/\* dequeueNode deletes node at tail of list

\* Postcondition list altered by node removal

\*/

Node\* LinkedList::dequeueNode() {

Node\* oldPtr = tailPtr;

if ( tailPtr != NULL ) {//If the list is not empty

tailPtr = tailPtr->prevPtr;

if ( tailPtr != NULL ) tailPtr->nextPtr = NULL;

else headPtr = NULL;

count--;

return oldPtr;

}

return NULL;

}

/\* deleteList deletes entire list deallocating all

\* nodes' memory

\* Postcondition list has no nodes, head/tail pointers are NULL

\*/

void LinkedList::deleteList() {

Node\* nodePtr = headPtr;

while( nodePtr != NULL ) {

headPtr = headPtr->nextPtr;

delete nodePtr;

nodePtr = headPtr;

}

tailPtr = NULL;

count = 0;

}

/\* operator= does a deep copy

\* Parameter

\* l list to copy

\* Returns reference to new list

\* Postcondition invoking list has all nodes deallocated then new nodes

\* allocated to copy from parameter list l

\*/

LinkedList& LinkedList::operator=( const LinkedList& l ) {

// thisPtr goes through this' list, curPtr goes thru l's

Node\* thisPtr = NULL, \*curPtr = l.headPtr;

headPtr = NULL;

// first, delete original list so no memory leak

deleteList();

if ( l.headPtr != NULL ) { // if there's list to copy

curPtr = l.headPtr;

headPtr = new Node( \*(l.headPtr) );

thisPtr = headPtr;

curPtr = curPtr->nextPtr;

while ( curPtr != NULL ) {

thisPtr->nextPtr = new Node( \*curPtr );

thisPtr->nextPtr->prevPtr = thisPtr;

thisPtr = thisPtr->nextPtr;

curPtr = curPtr->nextPtr;

}

}

tailPtr = thisPtr;

count = l.count;

return \*this;

}

8b (3 points)

**[PASTE THE SCREEN SHOT OF THE TESTS WITH A WHITE BACKGROUND FOR PAIR PROGRAMMIG 8B HERE]**

Text

Description automatically generated

**[PASTE CODE FOR PAIR PROGRAMMING 8B HERE – NOT A SCREEN SHOT. PASTE book.h AND book.cpp. HERE. MAKE SURE THE CODE IS SINGLE SPACED AND USE A COURIER NEW FONT]**

**Book.h:**

#ifndef BOOK\_H

#define BOOK\_H

#include <string>

#include <iostream>

//#include "node.h"

using namespace std;

class Book {

private:

string isbn;

string title;

string author;

string publisher;

int year;

int edition;

public:

// constructors

Book();

Book( string, string, string, string, int, int );

// copy constructor

Book( const Book& );

// destructor

virtual ~Book(){}

// set/get for private data

void setIsbn( string );

string getIsbn() const { return isbn; }

void setTitle( string );

string getTitle() const { return title; }

void setAuthor( string );

string getAuthor() const { return author; }

void setPublisher( string );

string getPublisher() const { return publisher; }

void setYear( int );

int getYear() const { return year; }

void setEdition( int );

int getEdition() const { return edition; }

void setBook( const Book& );

Book getBook() const { return \*this; }

// overloaded assignment operator

Book& operator=( const Book& );

// overloaded << and >>

friend ostream& operator<<( ostream&, const Book& );

friend istream& operator>>( istream&, Book& );

// overloaded ==

friend bool operator==( const Book&, const Book& );

// make Node class a friend

friend class Node;

//pp8b:

ostream& write(ostream&) const;

istream& read(istream&);

};

#endif

**book.cpp:**

/\* File: book.cpp

\* Author: Cindy

\* Description: implements a Book class

\*/

#include <string>

#include <iostream>

#include "book.h"

using namespace std;

/\* default constructor

\*/

Book::Book() {

year = edition = 0;

isbn = author = title = publisher = "";

}

/\* 6-arg constructor

\* Parameters:

\* i ISBN

\* t title

\* a author

\* p publisher

\* y year published

\* e edition

\*/

Book::Book( string i, string t, string a, string p, int y, int e ) {

setIsbn( i );

setTitle( t );

setAuthor( a );

setPublisher( p );

setYear( y );

setEdition( e );

}

/\* copy constructor

\*/

Book::Book( const Book& b ) {

setIsbn( b.isbn );

setTitle( b.title );

setAuthor( b.author );

setPublisher( b.publisher );

setYear( b.year );

setEdition( b.edition );

}

/\* setIsbn sets ISBN number

\* Parameter ISBN number

\* Returns: nothing

\*/

void Book::setIsbn( string i ) {

if ( i != "" ) isbn = i;

}

/\* setTitle sets book title

\* Parameter the title

\* Returns: nothing

\*/

void Book::setTitle( string t ) {

if ( t != "" ) title = t;

}

/\* setAuthor sets book's author

\* Parameters author name

\* Returns: nothing

\*/

void Book::setAuthor( string a ) {

if ( a != "" ) author = a;

}

/\* setPublisher sets book publisher

\* Parameters publisher name

\* Returns: nothing

\*/

void Book::setPublisher( string p ) {

if ( p != "" ) publisher = p;

}

/\* setYear sets year book was published

\* Parameters year

\*/

void Book::setYear( int y ) {

if ( y > 0 ) year = y;

}

/\* setEdition sets the book's edition

\* Parameters edition

\* Returns: nothing

\*/

void Book::setEdition( int e ) {

if ( e > 0 ) edition = e;

}

/\* setBook sets all of this Book object's data from

\* date in parameter Book object

\* Parameter book object to copy to this book object

\* Returns: nothing

\*/

void Book::setBook( const Book& b) {

isbn = b.isbn;

title = b.title;

author = b.author;

publisher = b.publisher;

year = b.year;

edition = b.edition;

}

/\* operator<< overloaded for Book object

\* Parameters:

\* outStream the output stream

\* b the book

\* Returns the output stream

\*/

ostream& operator<<( ostream& outStream, const Book& b ) {

outStream << b.isbn << " " << b.title << "\n\t" <<

b.author << "\n\t" << b.publisher << " " << b.year <<

" " << b.edition;

return outStream;

}

/\* operator>> overloaded for Book object

\* Parameters:

\* inStream the input stream

\* b the book

\* Returns the input stream

\*/

istream& operator>>( istream& inStream, Book& b ) {

getline( inStream, b.isbn );

getline( inStream, b.title );

getline( inStream, b.author );

getline( inStream, b.publisher );

inStream >> b.year >> b.edition;

return inStream;

}

/\* overloaded == operator returns true if books' ISBN's are equal

\* Parameters: two books to compare

\* Returns true if books' titles, and ISBNs are equal

\*/

bool operator==( const Book& b1, const Book& b2 ) {

return ( b1.title == b2.title && b1.isbn == b2.isbn );

}

/\* overloaded assignment operator

\* Parameter book on right-hand side of =

\* Returns: the book on the left-hand side of the =

\*/

Book& Book::operator=( const Book& b ) {

if ( this != &b ) { // don't copy the same object to itself

isbn = b.isbn;

title = b.title;

author = b.author;

publisher = b.publisher;

year = b.year;

edition = b.edition;

}

return \*this;

}

ostream& Book::write( ostream& outBin ) const {

char temp[128];

strcpy( temp, isbn.c\_str() );

outBin.write( temp, 18 ); //ISBN are 13 chars+4 dashes+\0

strcpy( temp, title.c\_str() );

outBin.write( temp, 128 ); // max title, etc. 127 chars

strcpy( temp, author.c\_str() );

outBin.write( temp, 128 );

strcpy( temp, publisher.c\_str() );

outBin.write( temp, 128 );

outBin.write( (char\*)&year, sizeof(int) );

outBin.write( (char\*)&edition, sizeof(int) );

return outBin;

}

istream& Book::read( istream& inBin ) {

char temp[128];

inBin.read(temp, 18);

isbn = temp;

inBin.read(temp, 128);

author = temp;

inBin.read(temp, 128);

publisher = temp;

inBin.read(temp, 128);

inBin.read((char\*)&year, sizeof(int));

inBin.read((char\*)&edition, sizeof(int));

return inBin;

}