**November 2014**

CSC 331 – DataBoss 1.0

Sam Jentsch/James Francis

Account Numbers: CSC331119/CSC331112

08

**Fall**

­­

Table of Contents

Introduction 3

User Guide 3

Display 3

Add 4

Delete – Primary 5

Delete - Secondary 6

Modify 6

Search - Primary 7

Search - Secondary 8

Search - Data 8

Help 8

Summary 10

Design Effort 10

Time Table 10

Implementation Effort 10

Problems 10

References 10

Appendix A – UML 11

Program Listing 13

# Introduction

First, thank you for choosing DataBoss as your record management solution. DataBoss will maintain your records with ease with the following commands:

* Display
* Add
* Delete\*
* Modify
* Search\*

\*work on both primary and secondary index lists

# User Guide

In order to run DataBoss with ease easy to understand commands, with appropriate user feedback, were implemented. All that you need to do is run: **DataBoss.sh** and the application will guide you through the process.

The executable generated by this shell script is **DataBoss**.

On DataBoss’s initial run it will report that no working files are present, in order to create the working files the user will need to provide DataBoss with a valid data file path, example:

**../instr/prog8.dat**

Once a valid file path has been entered the user can then hit and **DataBoss** will create the appropriate working files (index files as well as binary files).

DataBoss will continue running until the **quit** command is entered and the enter key pressed to commit the command.

What follows are example commands and what you will see if you were to execute these commands with DataBoss. DataBoss is not case sensitive.

## Display

1. Type: **DISPLAY**
2. Hit the enter/return button

General form:  **DISPLAY**

Example:

**DISPLAY**

The output should then be:

**DISPLAY**

**KEY LIST**

**12165 5**

**12345 1**

**12382 3**

**12434 2**

**16541 6**

**21212 7**

**34186 4**

**41742 8**

**SECONDARY KEY LIST**

**18 34186**

**19 21212**

**21 12434 16541**

**30 12165**

**45 12345**

**55 41742**

**62 12382**

**NUMBER OF RECORDS: 8**

**12345 Item06 45 14.2**

**12434 Item04 21 17.3**

**12382 Item09 62 41.37**

**34186 Item25 18 17.75**

**12165 Item16 30 7.69**

**16541 Item12 21 9.99**

**21212 Itme31 19 8.35**

**41742 Item14 55 12.36**

## Add

To use the ADD command follow the Instructions below:

1. **ADD key description code cost**
2. Hit the enter/return button

General form: **ADD key description code cost**

Example:

**ADD 12346 Itom01 99 21.12**

**INSERTING RECORD 12346 ITOM01 99 21.12**

**Setting header to: 9 Number is 9**

Using the **DISPLAY** command would result in the following output:

**DISPLAY**

**KEY LIST**

**12165 5**

**12345 1**

**12346 9**

**12382 3**

**12434 2**

**16541 6**

**21212 7**

**34186 4**

**41742 8**

**SECONDARY KEY LIST**

**18 34186**

**19 21212**

**21 12434 16541**

**30 12165**

**45 12345**

**55 41742**

**62 12382**

**99 12346**

**NUMBER OF RECORDS: 9**

**12345 Item06 45 14.2**

**12434 Item04 21 17.3**

**12382 Item09 62 41.37**

**34186 Item25 18 17.75**

**12165 Item16 30 7.69**

**16541 Item12 21 9.99**

**21212 Itme31 19 8.35**

**41742 Item14 55 12.36**

**12346 Itom01 99 21.12**

## Delete – Primary

To use the DELETE command (on the primary index) follow the Instructions below:

1. Enter: **DELETE PRIMARY** **key**
2. Hit the enter/return button

General form: **DELETE PRIMARY key**

Example:

**DELETE PRIMARY 41742**

The output should then be:

**DELETE PRIMARY**

**DELETE RECORD WITH CODE AND PRIMARY KEY: 55 41742**

Using the **DISPLAY** command would result in the following output:

**DISPLAY**

**KEY LIST**

**12165 5**

**12345 1**

**12346 9**

**12382 3**

**12434 2**

**16541 6**

**21212 7**

**34186 4**

**SECONDARY KEY LIST**

**18 34186**

**19 21212**

**21 12434 16541**

**30 12165**

**45 12345**

**62 12382**

**99 12346**

**NUMBER OF RECORDS: 8**

**12345 Item06 45 14.2**

**12434 Item04 21 17.3**

**12382 Item09 62 41.37**

**34186 Item25 18 17.75**

**12165 Item16 30 7.69**

**16541 Item12 21 9.99**

**21212 Itme31 19 8.35**

**-1 Item14 55 12.36**

**12346 Itom01 99 21.12**

## Delete - Secondary

To use the DELETE command (on the secondary index) follow the Instructions below:

1. Then enter: **DELETE SECONDARY code**
2. Hit the enter/return button

General form: **DELETE SECONDARY code**

Example:

**DELETE SECONDARY 21**

The output should then be:

**DELETE SECONDARY**

**DELETE WITH CODE: 21**

Using the **DISPLAY** command would yield in the following output:

**DISPLAY**

**KEY LIST**

**12165 5**

**12345 1**

**12382 3**

**21212 7**

**34186 4**

**41742 8**

**SECONDARY KEY LIST**

**18 34186**

**19 21212**

**30 12165**

**45 12345**

**55 41742**

**62 12382**

**NUMBER OF RECORDS: 6**

**12345 Item06 45 14.2**

**-1 Item04 21 17.3**

**12382 Item09 62 41.37**

**34186 Item25 18 17.75**

**12165 Item16 30 7.69**

**-1 Item12 21 9.99**

**21212 Itme31 19 8.35**

**41742 Item14 55 12.36**

## Modify

In order to use MODIFY properly a primary key must be supplied. Any modifications done to a primary or a secondary key must be made through the use of the primary key associated with a record.

1. Enter: **MOD key datalabel newdata**
   1. Where **key** is the primary key associated with the information you would like to change.
   2. Where **datalabel** represents the field within the record you would like to modify
      1. Possible options are: **KEY**, **DESCRIPTION**, **CODE**, or **COST**
   3. Where **newdata** represents the data that will be written to the record
      1. The information passed must be valid for the supplied field
2. Hit the enter/return button

General form: **MOD key datalabel newdata**

Example:

**MOD 21212 cost 00.00**

The output should then be:

**Inserted: 21212 Itme31 19 00.00**

Using the **Beta.o DISPLAY** command would yield in the following output:

**KEY LIST:**

**12165 5**

**12345 1**

**12382 3**

**12434 2**

**16541 6**

**21212 7**

**34186 4**

**41742 8**

**SECONDARY KEY LIST**

**18 34186**

**19 21212**

**21 12434 16541**

**30 12165**

**45 12345**

**55 41742**

**62 12382**

**Data Records:**

**12345 Item06 45 14.2**

**12434 Item04 21 17.3**

**12382 Item09 62 41.37**

**34186 Item25 18 17.75**

**12165 Item16 30 7.69**

**16541 Item12 21 9.99**

**21212 Itme31 19 00.0**

**41742 Item14 55 12.36**

## Search - Primary

To use the **Search** command (on the primary index) follow the Instructions below:

1. Then enter: **SEARCH PRIMARY key**
2. Hit the enter/return button

General form: **SEARCH PRIMARY key**

Example:

**SEARCH PRIMARY 12345**

The output should then be:

**SEARCH PRIMARY**

**12345  Item06 45  14.2**

## Search - Secondary

To use the **Search** command (on the secondar index) follow the Instructions below:

1. Enter: **SEARCH SECONDARY code**
2. Hit the enter/return button

General form: **SEARCH SECONDARY code**

Example:

**SEARCH SECONDARY 21**

The output should then be:

**SEARCH SECONDARY**

**RECORDS WITH CODE: 21**

**12434 Item04 21 17.3**

**16541 Item12 21 9.99**

## Search - Data

To use the **Search** command (on the data file) follow the Instructions below:

1. Then enter: **SEARCH DATA dataValue** 
   1. dataValue can be a **description** or a **cost**
2. Hit the enter/return button

General form: **SEARCH DATA dataValue**

Example:

**SEARCH DATA 41.37**

The output should then be:

**SEARCH DATA FILE**

**FOUND RECORD WITH COST: 41.37**

**12382 Item09 62 41.37**

## Help

To use the **Help** command follow the Instructions below:

1. Enter: **Help**
2. Hit the enter/return button

General form: **Help**

Example:

**Help**

The output should then be:

**-------------------------------------------------------------------------------------**

**Available Commands:**

**DISPLAY - Displays the contents of the file and the key lists.**

**QUIT - Close the files and exit DataBoss.**

**SEARCH:**

**Search takes a flag parameter followed by a data value.**

**The flag is either PRIMARY, SECONDARY, or DATA indicating the type of search you want to perform.**

**For example:**

**SEARCH PRIMARY 12345**

**SEARCH SECONDARY 99**

**SEARCH DATA Item99**

**DELETE:**

**Delete takes a flag parameter followed by a data value.**

**The flag is either PRIMARY or SECONDARY indicating the type of deletion you want to perform.**

**For example:**

**DELETE PRIMARY 12345**

**DELETE SECONDARY 99**

**MOD:**

**MOD takes a primary key to modify, the data label you wish to modify, and**

**the new value for the data as parameters.**

**The data label is either KEY, DESCRIPTION, CODE, or COST.**

**For example:**

**MOD 11111 KEY 22222**

**MOD 12345 DESCRIPTION Item11**

**MOD 11111 CODE 11**

**MOD 11111 COST 99.9**

**ADD:**

**ADD creates a new record and adds it to the system.**

**ADD is followed by a primary key, description, code, and cost: ADD PRIMARY\_KEY DESCRIPTION CODE COST**

**For example:**

**ADD 99999 Item99 99 99.9**

**-------------------------------------------------------------------------------------**

# Summary

## Design Effort

Object Oriented Programming methodologies were the focus of the design for DataBoss. The designs proposed in the project proposal were made with a modular zeal. During the implementation and testing phases of the Alpha and Beta releases of DataBoss we found that some of the classes and methods proposed did not need to exist, for want of coupling and cohesion.

UML Diagrams of classes and methods implemented in the final release of DataBoss can be found in Appendix A.

## Time Table

The incremental release of DataBoss met all deadlines established with the proposal of the project. No adjustments have been made to our timeline from the proposal’s timeline.

**Design** - 10/28/14 – *Delivered*

**Implementation**

Alpha *- 11/6/14 – Delivered* (Estimated time: 10 man hours)

Beta - 11/14/14 – *Delivered* (Estimated time: 10 man hours)

Release: 1.0 - *Delivered* - (Estimated time: 10 man hours)

**Testing** - *Complete* - (Estimated time: 6 man hours)

## Implementation Effort

Initial implementation efforts for DataBoss presented unique challenges for both team members. However after a few discussions about design and individual strengths and weaknesses regarding our abilities we came up with a compromise that would enable us to produce the best final product possible, in a timely manner, while allowing both team members to contribute to the project as a whole.

## Problems

Some problems that occurred were mostly due to oversights on our part. We either assumed that current functionality (or lack thereof) was sufficient for prior releases, or overlooked some things all together such as not looping the program until the user wanted to quit the application. Error handling proved to be the most challenging problem that we encountered. Particularly dealing with erroneous input passed to the modify command. During the testing process, if bad input was given to the modify command more than twice the program would terminate with an lldb error, a vector would be thrown out of bounds. The problem was found in the logic for selecting commands and resolved after additional testing.

For the scope of this project DataBoss appears to have no significant limitations, with regards to its intended purpose.

# References

Stack Overflow

How to Program C++ (9th Ed. Intl.): Deitel & Deitel

# Appendix A – UML

# Program Listing

InputManager.cpp

/\*

---------------------------------------------------------------------------------

PROGRAM NAME: DataBoss

PROGRAMMER: Samuel Jentsch

CLASS: CSC 331.001, Fall 2014

INSTRUCTOR: Dr. R. Strader

REFERENCES: C++ How to Program

Paul Deitel & Harvey Deitel

Dr. Strader: assignment information sheet

PROGRAM PURPOSE:

DataBoss is a single file database manager.

It is capable of the following commands allowing for modification and

access to the file:

DISPLAY - Displays the contents of the file and the key lists.

QUIT - Close the files and exit DataBoss.

SEARCH:

Search takes a flag parameter followed by a data value.

The flag is either PRIMARY, SECONDARY, or DATA indicating the

type of search you want to perform.

For example:

SEARCH PRIMARY 12345

SEARCH SECONDARY 99

SEARCH DATA Item99

DELETE:

Delete takes a flag parameter followed by a data value.

The flag is either PRIMARY or SECONDARY indicating the type of

deletion you want to perform.

For example:

DELETE PRIMARY 12345

DELETE SECONDARY 99

MOD:

MOD takes a primary key to modify, the data label you wish to modify, and

the new value for the data as parameters.

The data label is either KEY, DESCRIPTION, CODE, or COST.

For example:

MOD 11111 KEY 22222

MOD 12345 DESCRIPTION Item11

MOD 11111 CODE 11

MOD 11111 COST 99.9

ADD:

ADD creates a new record and adds it to the system.

ADD is followed by a primary key, description, code, and cost:

ADD PRIMARY\_KEY DESCRIPTION CODE COST

For example:

ADD 99999 Item99 99 99.9

VARIABLE DICTIONARY:

indexManager - IndexListManager, the main instance of the IndexListManager class

used throughout the program while in operation. Responsible for

all actions associated with both the primary and secondary key

list.

productManager - ProductManager, the main instance of the ProductManager class

used throughout the program while in operation. Responsbile for

handling all interaction with the binaryDataFile.

textDataFile - ifstream, a stream containing a reference to the file

used to initialize the data file (the original text file).

primaryIndexFile - fstream, a stream allowing for input and output to a binary

file containing the primary key list (maintained in

IndexListManager)

secondaryIndexFile - fstream, a stream allowing for input and output to a binary

file containing the secondary key list (maintained in

IndexListManager)

binaryDataFile - fstream, a stream allowing for input and output to a binary file

containing the binary representation of the records present in

textDataFile. This file is used to manipulate and access the

records contained in the data file (maintained in ProductManager)

---------------------------------------------------------------------------------

\*/

#include <iostream>

#include <fstream>

#include "IndexListManager.h"

#include "ProductManager.h"

using namespace std;

IndexListManager \*indexManager;

ProductManager \*productManager;

ifstream textDataFile;

fstream primaryIndexFile;

fstream secondaryIndexFile;

fstream binaryDataFile;

bool parseInput(vector<string> arguments);

void handleHelp();

void handleSearchForPrimaryKey(int key);

void handleSearchForSecondaryKey(int key);

void handleSearchForDataValue(string dataValue);

void handleAdd(string record);

void handleDeletePrimary(int key);

void handleDeleteSecondary(int code);

void handleModify(int key, string dataLabel, string newData);

int getFlagForDataLabel(string dataLabel);

string toUpperCase(string lowerCaseString);

bool checkDouble(string checkString);

int convertStringToInt(std::string intString);

double convertStringToDouble(std::string doubleString);

void welcomeMessage();

void createFiles();

int main(int argc, const char \* argv[]) {

productManager = new ProductManager(binaryDataFile);

indexManager = new IndexListManager(primaryIndexFile, secondaryIndexFile);

primaryIndexFile.open("PrimaryKey.ixp", fstream::ate |fstream::out | fstream::in | fstream::binary);

secondaryIndexFile.open("SecondaryKey.ixp", fstream::ate |fstream::out | fstream::in | fstream::binary);

binaryDataFile.open("BinaryDataFile.dat", fstream::ate | fstream::binary| fstream::out | fstream::in);

welcomeMessage();

if (!binaryDataFile.is\_open() || !primaryIndexFile.is\_open() || !secondaryIndexFile.is\_open()) {

cout << "No files currently exist. Please enter a path to a data file: ";

createFiles();

} else {

cout << "There are currently files available for modification." << endl;

cout << "Would you like to use these files? (Y/N): ";

string response;

cin >> response;

response[0] = toupper(response[0]);

if (response.compare("N") == 0) {

cout << "DataBoss will create new files. Please enter a path to the data file: ";

createFiles();

} else {

cout << "DataBoss will operate on the existing files." << endl;

indexManager->populateKeyListFromIndexRecordFile();

indexManager->populateSecondaryKeyListFromFile();

}

}

bool continueProgram = true;

string line;

getline(cin, line);

while (continueProgram) {

//Get input from user.

cout << "DataBoss: ";

getline(cin, line);

istringstream commands(line);

string command;

vector<string> commandArray;

while (!commands.eof()) {

commands >> command;

commandArray.push\_back(command);

}

cout << "------------------------------------------------------------------------------------------------------" << endl;

continueProgram = parseInput(commandArray);

cout << "------------------------------------------------------------------------------------------------------" << endl;

commands.clear();

}

binaryDataFile.flush();

binaryDataFile.close();

primaryIndexFile.flush();

primaryIndexFile.close();

return 0;

}

void createFiles() {

//--------------------------------------------------------------------------------//

//Get user input for the path to the data file. Use this path to open and a ifstream

//to the data file.

//Initialize the files (global variables in the program). Create the files and open

//them so they are ready to pass to the productManager and indexManager to initialize

//the binary representation of the text file and the primary/secondary index lists.

//Preconditions: Global variables for primaryIndexFile, secondaryIndexFile, and

//binaryDataFile. Additionally, references to productManager and indexManager must

//be available to initialize the binary file and index lists.

//Postconditions: Based on the user input, the file is opened, new files for the

//binary data file, primary index list, and secondary index lists are created.

//--------------------------------------------------------------------------------//

//Open the text data file

primaryIndexFile.close();

secondaryIndexFile.close();

binaryDataFile.close();

string textFilePath;

cin >> textFilePath;

textDataFile.open(textFilePath.c\_str(), ios::in);

while (!textDataFile.is\_open()) {

//File not found.

cout << "A file with that path was not found. Please enter a new path to data file: ";

cin >> textFilePath;

textDataFile.open(textFilePath.c\_str(), ios::in);

}

cout << "CREATING NEW FILES" << endl;

ofstream binCreate("BinaryDataFile.dat", fstream::out);

binCreate.close();

binaryDataFile.open("BinaryDataFile.dat", fstream::binary| fstream::out | fstream::in);

productManager->createBinaryRecordFile(textDataFile);

textDataFile.close();

ofstream indexCreate("PrimaryKey.ixp", fstream::out);

indexCreate.close();

ofstream secondaryCreate("SecondaryKey.ixp", fstream::out);

secondaryCreate.close();

secondaryIndexFile.open("SecondaryKey.ixp", fstream::out | fstream::in | fstream::binary);

primaryIndexFile.open("PrimaryKey.ixp", fstream::binary| fstream::out | fstream::in);

indexManager->populateKeyListFromDataRecordFile(binaryDataFile);

indexManager->printKeyList();

indexManager->savePrimaryKeyListToFile();

indexManager->saveSecondaryKeyListToFile();

}

void welcomeMessage() {

//--------------------------------------------------------------------------------//

//Display a fun, unique, endearing, lovable, cute, sweet, charming, appealing,

//attractive, winning, captivating, enchanting, beguiling, winsome, kawaii,

//engaging welcome message to the user!

//Preconditions: None

//Postconditions: Welcome message is displayed to the user.

//--------------------------------------------------------------------------------//

cout << " \_ .-') \_ ('-. .-') \_ ('-. \n( ( OO) ) ( OO ).-.( OO) ) ( OO ).-. \n \\ .'\_ / . --. // '.\_ / . --. / \n ,`'--...\_) | \\-. \\ |'--...\_\_) | \\-. \\ \n | | \\ '.-'-' | |'--. .--'.-'-' | | \n | | ' | \\| |\_.' | | | \\| |\_.' | \n | | / : | .-. | | | | .-. | \n | '--' / | | | | | | | | | | \n `-------' `--' `--' `--' `--' `--' \n.-. .-') .-') .-') \n\\ ( OO ) ( OO ). ( OO ). \n ;-----.\\ .-'),-----. (\_)---\\\_)(\_)---\\\_) \n | .-. | ( OO' .-. '/ \_ | / \_ | \n | '-' /\_)/ | | | |\\ :` `. \\ :` `. \n | .-. `. \\\_) | |\\| | '..`''.) '..`''.) \n | | \\ | \\ | | | |.-.\_) \\.-.\_) \\ \n | '--' / `' '-' '\\ /\\ / \n `------' `-----' `-----' `-----' " << endl;

cout << "Welcome to DataBoss!\n";

cout << "Type HELP for a list of available commands." << endl;

cout << "Type QUIT to save and exit." << endl;

}

bool parseInput(vector<string> arguments) {

//--------------------------------------------------------------------------------//

//Parse the input passed and execute the appropriate command. Error checks are

//performed on the input to ensure the command is valid, the correct amount of

//arguments are present, and the arguments are the correct type.

//Preconditions: vector of strings containing the command arguments passed as a

//parameter.

//Postconditions: the arguments are parsed, checked for errors, and executed. Either

//way a message indicating what happened is displayed to the user.

//--------------------------------------------------------------------------------//

bool executedCommand = false;

int numberOfArguments = (int)arguments.size();

string command(arguments[0]);

command = toUpperCase(command);

if (numberOfArguments == 1) {

//Display

if (command.compare("DISPLAY") == 0) {

cout << "DISPLAY" << endl;

indexManager->printKeyList();

productManager->traverseFile();

executedCommand = true;

} else if (command.compare("HELP") == 0) {

handleHelp();

executedCommand = true;

} else if (command.compare("QUIT") == 0) {

cout << "Closing Files and quitting the program... Please come again soon!" << endl;

return false;

}

}

else if (numberOfArguments == 3) {

string flag = toUpperCase(arguments[1]);

string keyString = arguments[2];

//DELETE

if (command.compare("DELETE") == 0) {

if (flag.compare("PRIMARY") == 0) {

//Delete Primary

cout << "DELETE PRIMARY" << endl;

executedCommand = true;

handleDeletePrimary(convertStringToInt(keyString));

} else if (flag.compare("SECONDARY") == 0) {

//Delete Secondary

cout << "DELETE SECONDARY" << endl;

executedCommand = true;

handleDeleteSecondary(convertStringToInt(keyString));

}

}//end if delete

//SEARCH

else if(command.compare("SEARCH") == 0) {

if (flag.compare("PRIMARY") == 0) {

//SEARCH Primary

cout << "SEARCH PRIMARY" << endl;

executedCommand = true;

handleSearchForPrimaryKey(convertStringToInt(keyString));

} else if (flag.compare("SECONDARY") == 0) {

//SEARCH Secondary

cout << "SEARCH SECONDARY" << endl;

executedCommand = true;

handleSearchForSecondaryKey(convertStringToInt(keyString));

} else if (flag.compare("DATA") == 0) {

//SEARCH Data File

cout << "SEARCH DATA FILE" << endl;

executedCommand = true;

handleSearchForDataValue(keyString);

}

}

}

else if (numberOfArguments == 4) {

//MOD KEY DATA\_LABEL NEW\_DATA

if (command.compare("MOD") == 0) {

cout << "MOD" << endl;

executedCommand = true;

string keyString = arguments[1];

string dataLabel = toUpperCase(arguments[2]);

string newData = arguments[3];

handleModify(convertStringToInt(keyString), dataLabel, newData);

}

}

else if (numberOfArguments == 5) {

//ADD KEY NAME CODE COST

if (command.compare("ADD") == 0) {

cout << "ADD" << endl;

executedCommand = true;

string record;

record.append(arguments[1]);

record.append(" ");

record.append(arguments[2]);

record.append(" ");

record.append(arguments[3]);

record.append(" ");

record.append(arguments[4]);

handleAdd(record);

}

}

if (!executedCommand) {

cout << "Invalid command. Type HELP for a list of available commands." << endl;

}

return true;

}

void handleHelp() {

//--------------------------------------------------------------------------------//

//Display a list of available commands to the user along with a description of the

//command and examples of its usage.

//Preconditions: None

//Postconditions: the help message is displayed to the console.

//--------------------------------------------------------------------------------//

cout << "Available Commands: " << endl;

cout << "DISPLAY - Displays the contents of the file and the key lists." << endl;

cout << "QUIT - Close the files and exit DataBoss." << endl;

cout << "SEARCH:\n";

cout << "\tSearch takes a flag parameter followed by a data value.\n";

cout << "\tThe flag is either PRIMARY, SECONDARY, or DATA indicating the type of search you want to perform." << endl;

cout << "\tFor example:";

cout << "\n\t\tSEARCH PRIMARY 12345\n\t\tSEARCH SECONDARY 99\n\t\tSEARCH DATA Item99" << endl;

cout << "DELETE:\n";

cout << "\tDelete takes a flag parameter followed by a data value.\n";

cout << "\tThe flag is either PRIMARY or SECONDARY indicating the type of deletion you want to perform." << endl;

cout << "\tFor example:";

cout << "\n\t\tDELETE PRIMARY 12345\n\t\tDELETE SECONDARY 99" << endl;

cout << "MOD:\n";

cout << "\tMOD takes a primary key to modify, the data label you wish to modify, and \n\tthe new value for the data as parameters.\n";

cout << "\tThe data label is either KEY, DESCRIPTION, CODE, or COST." << endl;

cout << "\tFor example:";

cout << "\n\t\tMOD 11111 KEY 22222\n\t\tMOD 12345 DESCRIPTION Item11\n\t\tMOD 11111 CODE 11\n\t\tMOD 11111 COST 99.9" << endl;

cout << "ADD:\n";

cout << "\tADD creates a new record and adds it to the system.\n";

cout << "\tADD is followed by a primary key, description, code, and cost: ADD PRIMARY\_KEY DESCRIPTION CODE COST" << endl;

cout << "\tFor example:";

cout << "\n\t\t ADD 99999 Item99 99 99.9" << endl;

}

void handleSearchForPrimaryKey(int key) {

//--------------------------------------------------------------------------------//

//Handle the search for a primary key matching the key parameter passed.

//Preconditions: int key passed as parameter to search for.

//Postconditions: the primary key list is searched for the primary key passed if it

//is found, the record at the RRN corresponding to the primary key is gotten and

//displayed to the console . Otherwise, a message that it was not found is displayed.

//--------------------------------------------------------------------------------//

//Offset is -1 if record not found in key list.

int offset = indexManager->getRecordOffsetForKey(key);

if (offset == -1) {

cout << "RECORD NOT FOUND IN SEARCH." << endl;

} else {

DataRecord recordRead = productManager->getDataRecordAtOffset(offset);

cout << "FOUND: ";

recordRead.printRecord();

}

}

void handleSearchForSecondaryKey(int key) {

//--------------------------------------------------------------------------------//

//Handle the search for a secondary key matching the key parameter passed.

//Preconditions: int key passed as parameter to search for.

//Postconditions: the secondary key list is searched for the secondary key passed.

//If it is found, the records at the RRNs corresponding to the code are gotten and

//displayed to the console . Otherwise, a message that it was not found is displayed.

//--------------------------------------------------------------------------------//

vector<int> primaryKeys = indexManager->getPrimaryKeysWithCode(key);

if (primaryKeys.size() == 0) {

cout << "SECONDARY KEY NOT FOUND IN SEARCH." << endl;

return;

}

int RRN = -1;

DataRecord record;

cout << "RECORDS WITH CODE: " << key << endl;

for (int i = 0; i < primaryKeys.size(); i++) {

RRN = indexManager->getRecordOffsetForKey(primaryKeys.at(i));

record = productManager->getDataRecordAtOffset(RRN);

record.printRecord();

}

}

void handleSearchForDataValue(string data) {

//--------------------------------------------------------------------------------//

//Handle the search for a data value matching the key parameter passed. It is

//determined if the value passed is a double or string and the appropriate field is

//searched based on the result (cost for double, description for string).

//Preconditions: string representing data value passed as a parameter.

//Postconditions: the data file is searched for the data value passed.

//If it is found, the record containing the data value is displayed to the console.

//Otherwise, a message indicating that the data value couldn't be found is displayed.

//--------------------------------------------------------------------------------//

//Description OR Cost

bool isDouble = checkDouble(data);

if (isDouble) {

//Handle search of double value cost

double cost = convertStringToDouble(data);

DataRecord foundRecord = productManager->searchFileForCost(cost);

if (foundRecord.getKey() != -1) {

cout << "FOUND RECORD WITH COST: " << cost << endl;

foundRecord.printRecord();

} else {

cout << "COULD NOT FIND RECORD WITH COST: " << cost << endl;

}

} else {

//Handle search for description

DataRecord foundRecord = productManager->searchFileForDescription(data);

if (foundRecord.getKey() != -1) {

cout << "FOUND RECORD WITH NAME: " << data << endl;

foundRecord.printRecord();

} else {

cout << "COULD NOT FIND RECORD WITH NAME: " << data << endl;

}

}

}

void handleAdd(string record) {

//--------------------------------------------------------------------------------//

//Handle the addition of a new data record represented by the string parameter passed.

//The string is converted into a data record using the getDataRecordForString method

//in the ProductManager class. This method will display a message to the console

//and return a record with -1 in its key field if an error occurred when converting

//any part of the record string into a data record field.

//Preconditions: a string representing a record passed as a parameter.

//Postconditions: the string passed is converted into a data record and added to the

//file, primary index list, and secondary index list.

//If an error occurred while converting the string into a data record a message is

//displayed to the user and the record is not created.

//--------------------------------------------------------------------------------//

DataRecord newRecord = productManager->getDataRecordForString(record);

if (newRecord.getKey() == -1) {

cout << "Invalid parameter type(s) for ADD. Please try again. (See HELP for syntax)" << endl;

} else {

int RRN = productManager->addDataRecordToBinaryFile(newRecord);

indexManager->addDataRecordToKeyList(newRecord, RRN);

indexManager->addDataRecordToInvertedList(newRecord);

indexManager->savePrimaryKeyListToFile();

indexManager->saveSecondaryKeyListToFile();

}

}

void handleDeletePrimary(int key) {

//--------------------------------------------------------------------------------//

//Handle the deletion of a primary key. This includes deleting the record containing

//the primary key from the data file, primary index list, and secondary index list.

//Preconditions: int key passed as parameter to delete.

//Postconditions: the record matching the key passed is searched for, and if it is

//found, removed from the record file (a -1 is put in its key field and written to

//its original location), the primary index list, and secondary index list.

//If a record with the key passed could not be found a message is displayed to the

//user indicating the failure.

//--------------------------------------------------------------------------------//

int offset = indexManager->getRecordOffsetForKey(key);

if (offset == -1) {

cout << "RECORD NOT FOUND TO DELETE." << endl;

} else {

DataRecord deleteRecord = productManager->getDataRecordAtOffset(offset);

//Delete Secondary Index Entry for Primary Key deletion

int code = deleteRecord.getCode();

int primaryKey = deleteRecord.getKey();

cout << "DELETE RECORD WITH CODE AND PRIMARY KEY: " << code << " " << primaryKey << endl;

indexManager->removePrimaryIndexFromSecondaryKeyWithCode(code, primaryKey);

//Delete Primary Index Entry for Primary Key deletion

indexManager->deleteIndexRecordWithKey(key);

//Delete Record Entry for Primary Key deletion

productManager->deleteRecordAtOffset(offset);

indexManager->savePrimaryKeyListToFile();

indexManager->saveSecondaryKeyListToFile();

}

}

void handleDeleteSecondary(int code) {

//--------------------------------------------------------------------------------//

//Handle the deletion for a secondary key matching the key parameter passed.

//Preconditions: int code passed as parameter to delete

//Postconditions: the secondary key list is searched for the secondary key passed.

//If it is found, the entry is removed from the secondary key list, but the data

//record remains unchanged.

//--------------------------------------------------------------------------------//

int found = indexManager->deleteSecondaryIndexWithKey(code);

if (found != -1) {

indexManager->saveSecondaryKeyListToFile();

cout << "DELETED SECONDARY INDEX ENTRY FOR: " << code << endl;

} else {

cout << "COULD NOT FIND SECONDARY INDEX FOR CODE: " << code << endl;

}

}

void handleModify(int key, string dataLabel, string newData) {

//--------------------------------------------------------------------------------//

//Handle the modification for the primary key passed. Check the datalabel to retrieve

//a numeric flag for parsing. Check the newData string to ensure it matches the data

//required for the dataLabel being modified.

//Preconditions: int key passed as the primary key of the record to modify. A

//dataLabel string for a flag indicating what datafield should be altered. The new

//data to change the desired data field to.

//Postconditions: the record matching the primary key passed is retrieved. The

//datalabel is parsed and the newData is converted into the required data type.

//The retrieved record is altered with the new data in the field indicated by

//dataLabel. The data file is updated.

//If a code or primary key was altered the secondary index list and primary index

//list is updated accordingly.

//If there was an error when finding the record with the key, and invalid dataLabel

//flag, or converting the newData string into the requried data type, a message is

//displayed to console and the record, files, and lists remain unchanged.

//--------------------------------------------------------------------------------//

int dataLabelFlag = getFlagForDataLabel(dataLabel);

bool goodData = true;

int modifiedRecordOffset = indexManager->getRecordOffsetForKey(key);

DataRecord modifiedRecord;

if (modifiedRecordOffset == -1) {

cout << "RECORD NOT FOUND TO MODIFY." << endl;

} else {

modifiedRecord = productManager->getDataRecordAtOffset(modifiedRecordOffset);

}

if (dataLabelFlag != -1 && modifiedRecordOffset != -1) {

switch (dataLabelFlag) {

case 1:

//KEY modification

modifiedRecord.setKey(convertStringToInt(newData));

if (modifiedRecord.getKey() != -1) {

indexManager->removePrimaryIndexFromSecondaryKeyWithCode(modifiedRecord.getCode(), modifiedRecord.getKey());

indexManager->deleteIndexRecordWithKey(key);

indexManager->addDataRecordToKeyList(modifiedRecord, modifiedRecordOffset);

indexManager->addDataRecordToInvertedList(modifiedRecord);

indexManager->savePrimaryKeyListToFile();

indexManager->saveSecondaryKeyListToFile();

} else {

cout << "Invalid Primary Key. Must be integer.\n";

goodData = false;

}

break;

case 2:

//DESCRIPTION modification

if (newData.size() <= 8) {

modifiedRecord.setName(newData.c\_str());

} else {

std::cout << "Invalid Description. Can be up to 8 characters.\n";

goodData = false;

}

break;

case 3:

//CODE modification

//Affects secondary index list

if (convertStringToInt(newData) != -1) {

indexManager->removePrimaryIndexFromSecondaryKeyWithCode(modifiedRecord.getCode(), modifiedRecord.getKey());

modifiedRecord.setCode(convertStringToInt(newData));

indexManager->addDataRecordToInvertedList(modifiedRecord);

indexManager->saveSecondaryKeyListToFile();

} else {

std::cout << "Invalid Code. Must be integer value.\n";

goodData = false;

}

break;

case 4:

//COST modification

modifiedRecord.setCost(convertStringToDouble(newData));

if (modifiedRecord.getCost() < 0) {

std::cout << "Invalid Cost. Must be double value.\n";

goodData = false;

}

break;

//default:

// break;

}//end switch for data type

if (goodData) {

productManager->writeDataRecordToBinaryFile(modifiedRecord, modifiedRecordOffset);

} else {

cout << "Invalid parameter type for MOD. Please try again. (See HELP for syntax)" << endl;

}

}//end else

}//end handle modify

int getFlagForDataLabel(string dataLabel) {

//--------------------------------------------------------------------------------//

//Return a number for the datalabel passed to facilitate parsing of the modify

//command. A number representing a specific data type is returned.

//Preconditions: dataLabel string passed as parameter to match and return an number

//for.

//Postconditions: the string is checked against the data type and a number is

//returned. If the dataLabel didn't match a supported data flag, -1 is returned to

//the caller.

//--------------------------------------------------------------------------------//

int dataFlag = -1;

if (dataLabel.compare("KEY") == 0) {

dataFlag = 1;

} else if (dataLabel.compare("DESCRIPTION") == 0) {

dataFlag = 2;

} else if (dataLabel.compare("CODE") == 0) {

dataFlag = 3;

} else if (dataLabel.compare("COST") == 0) {

dataFlag = 4;

}

return dataFlag;

}

#pragma mark - Conversion Methods For Parsing

bool checkDouble(string checkString) {

//--------------------------------------------------------------------------------//

//Check if the string passed is a double value.

//Preconditions: string to check passed as parameter.

//Postconditions: true is returned if the string is a double, false if not.

//--------------------------------------------------------------------------------//

bool isDouble = true;

for (int i = 0; i < checkString.size() && isDouble; i++) {

if (checkString.at(i) != '.' && isalpha(checkString.at(i)))

isDouble = false;

}

return isDouble;

}

string toUpperCase(string lowerCaseString) {

//--------------------------------------------------------------------------------//

//Convert the string passed to uppercase and return to the caller.

//Preconditions: string to convert to uppercase passed as parameter.

//Postconditions: the string is converted to uppercase and returned.

//--------------------------------------------------------------------------------//

for (int i = 0; i < lowerCaseString.length(); i++) {

lowerCaseString[i] = toupper(lowerCaseString[i]);

}

return lowerCaseString;

}

int convertStringToInt(std::string intString) {

//--------------------------------------------------------------------------------//

//Convert the string passed into an integer and return the integer value.

//Preconditions: string representation of an integer passed as parameter.

//Postconditions: the string passed is parsed and the integer value of the string

//is returned to the caller.

//--------------------------------------------------------------------------------//

int convertedInteger;

std::istringstream converter(intString);

if (converter >> convertedInteger) {

return convertedInteger;

} else {

return -1;

}

}

double convertStringToDouble(std::string doubleString) {

//--------------------------------------------------------------------------------//

//Convert the string passed into a double and return the double value.

//Preconditions: string representation of a double passed as parameter.

//Postconditions: the string passed is parsed and the double value of the string

//is returned to the caller.

//--------------------------------------------------------------------------------//

double convertedDouble;

std::istringstream converter(doubleString);

if (converter >> convertedDouble) {

return convertedDouble;

} else {

return -1;

}

return convertedDouble;

}

ProductManager.h

/\*

---------------------------------------------------------------------------------

CLASS NAME: ProductManager

PROGRAMMER: Samuel Jentsch

CLASS: CSC 331.001, Fall 2014

INSTRUCTOR: Dr. R. Strader

REFERENCES: C++ How to Program

Paul Deitel & Harvey Deitel

Dr. Strader: assignment information sheet

CLASS PURPOSE:

Product Manager is responsible for all actions associated with the binary data

file, including:

- Creating the binary representation of the text data file.

- Inserting data records.

- Deleting data records.

- Retrieving records at a specified index.

- Converting a sting representing a Data Record into an instance of the Data

Record class for use.

- Traversing the data record file for display to the console.

- Searching the data record file for a specific data attribute.

- Maintaining the header of the data record file for the number of items.

VARIABLE DICTIONARY:

&binaryRecordFile, fstream - a reference to the binary data record file that will

be searched, inserted into, retrieved from, etc.

headerNumber - int, a global count of the number of binary records in the data

record file.

---------------------------------------------------------------------------------

\*/

#ifndef \_\_TermProject\_Alpha\_\_ProductManager\_\_

#define \_\_TermProject\_Alpha\_\_ProductManager\_\_

#include <stdio.h>

#include <sstream>

#include <string>

#include <vector>

#include <iostream>

#include <fstream>

#include "DataRecord.h"

class ProductManager {

private:

std::fstream &binaryRecordFile;

int headerNumber;

public:

ProductManager(std::fstream &binaryRecordFile);

//\*\*\*Data Record Binary File Reader\*\*\*//

DataRecord getDataRecordAtOffset(int offset);

int getNumberOfRecords();

//\*\*\*Data Record Binary File Writer\*\*\*//

void createBinaryRecordFile(std::ifstream &textFile);

void writeDataRecordToBinaryFile(DataRecord newRecord, int line);

void updateHeader(int numberOfRecords);

//\*\*\*Add\*\*\*//

int addDataRecordToBinaryFile(DataRecord record);

//\*\*\*Display\*\*\*//

void traverseFile();

//\*\*\*Delete\*\*\*//

void deleteRecordAtOffset(int offset);

//\*\*\*Search\*\*\*//

DataRecord searchFileForCost(double cost);

DataRecord searchFileForDescription(std::string description);

DataRecord getRecordWithKey(int key);

//\*\*\*Conversion Methods\*\*\*//

DataRecord getDataRecordForString(std::string record);

int convertStringToInt(std::string intString);

double convertStringToDouble(std::string doubleString);

std::vector<std::string> split(const std::string &s, char delim, std::vector<std::string> &elems);

};

#endif /\* defined(\_\_TermProject\_Alpha\_\_ProductManager\_\_) \*/

ProductManager.cpp

#include "ProductManager.h"

#pragma mark - Constructor

ProductManager::ProductManager(std::fstream &binaryRecordFile)

:binaryRecordFile(binaryRecordFile)

{

}

#pragma mark - Read From File

DataRecord ProductManager::getDataRecordAtOffset(int offset) {

//--------------------------------------------------------------------------------//

//Read in a data record from the data file at the offset passed. Return this record

//to the caller as an instance of the DataRecord class.

//Preconditions: an offset to read a record from binaryRecordFile class variable.

//Postconditions: an instance of DataRecord is instantiated using the data at the

//record passed and returned to the caller.

//--------------------------------------------------------------------------------//

binaryRecordFile.seekg(0);

binaryRecordFile.seekg(offset \* sizeof(DataRecord));

DataRecord record;

binaryRecordFile.read(reinterpret\_cast<char\*>(&record), sizeof(DataRecord));

return record;

}

void ProductManager::traverseFile() {

//--------------------------------------------------------------------------------//

//Traverse the entire binary record file from beginning to end. Display the contents

//of the file to the console, beginning with the header record indicating the number

//of records in the file (also used to determine the number of records to iterate

//through for printing).

//Preconditions: binaryRecordFile class variable with valid header record.

//Postconditions: the entire binaryRecordFile is traversed and displayed to the

//console.

//--------------------------------------------------------------------------------//

binaryRecordFile.seekg(0);

DataRecord header;

std::cout<<"DATA FILE: " << "\n";

int numberOfRecords = getNumberOfRecords();

std::cout << "NUMBER OF RECORDS: " << numberOfRecords <<"\n";

DataRecord record;

for (int i = 0; i < numberOfRecords; i++) {

//std::cout << "READING: ";

binaryRecordFile.read(reinterpret\_cast< char \* >(&record), sizeof(DataRecord));

record.printRecord();

}

}

int ProductManager::getNumberOfRecords() {

//--------------------------------------------------------------------------------//

//Seek to the beginning of the binary record file and read in the header record.

//The header record is read and converted into an integer. This integer is returned

//to the caller (the number of records in the binary file.

//Preconditions: binaryRecordFile class variable

//Postconditions: the header record is read from the file, converted to an integer,

//and returned to the caller.

//--------------------------------------------------------------------------------//

DataRecord header;

binaryRecordFile.seekg(0);

binaryRecordFile.read(reinterpret\_cast< char \* >(&header), sizeof(DataRecord));

int numberOfRecords = header.getKey();

headerNumber = numberOfRecords;

//std::cout << "SETTING HEADER NUMBER TO: " <<headerNumber << std::endl;

return numberOfRecords;

}

#pragma mark - Write To File

void ProductManager::createBinaryRecordFile(std::ifstream &textFile) {

//--------------------------------------------------------------------------------//

//Create a binary representation of the text record file passed.

//Preconditions: text file passed as parameter. Class variable binaryRecordFile

//pointing to an instance of fstream opened to a valid binary file.

//Postconditions: the text record file is traversed and each line is converted into

//an instance of the DataRecord class. This instance is written to the

//binaryRecordFile.

//--------------------------------------------------------------------------------//

int lines = 0;

std::string dataLine;

binaryRecordFile.seekp(0);

DataRecord newRecord;

while (getline(textFile, dataLine)) {

//Create duplicate binary file for seeking:

newRecord = getDataRecordForString(dataLine);

writeDataRecordToBinaryFile(newRecord, lines);

lines++;

}

}

void ProductManager::writeDataRecordToBinaryFile(DataRecord newRecord, int line) {

//--------------------------------------------------------------------------------//

//Duplicate the text record to a fixed field and write the record to the binary

//file.

//Preconditions: record string and line to write to passed as parameters.

//Postconditions: record string converted to data record and written to file.

//--------------------------------------------------------------------------------//

//DataRecord newRecord = getDataRecordForString(record);

std::cout << "INSERTING RECORD: ";

newRecord.printRecord();

int recordOffset = sizeof(DataRecord) \* line;

binaryRecordFile.seekp(recordOffset);

binaryRecordFile.write(reinterpret\_cast< char \* >(&newRecord), sizeof(DataRecord));

}

void ProductManager::updateHeader(int numberOfRecords) {

//--------------------------------------------------------------------------------//

//Update the header record of the binary record file.

//Preconditions: global headerNumber representing the previous number of records.

//Postconditions: header record is updated by incrementing the header number. The

//header is stored in the key field of a datarecord and written as the first record

//in the binary record file.

//--------------------------------------------------------------------------------//

DataRecord header;

header.setKey(++headerNumber);

std::cout<<"Setting header to: " << header.getKey() << " Header Number is: " << headerNumber << "\n";

binaryRecordFile.seekp(0);

binaryRecordFile.write(reinterpret\_cast< char \* >(&header), sizeof(DataRecord));

}

int ProductManager::addDataRecordToBinaryFile(DataRecord record) {

//--------------------------------------------------------------------------------//

//Update the header record of the binary record file.

//Preconditions: global headerNumber representing the previous number of records.

//Postconditions: header record is updated by incrementing the header number. The

//header is stored in the key field of a datarecord and written as the first record

//in the binary record file.

//--------------------------------------------------------------------------------//

int RRN = getNumberOfRecords() + 1;

writeDataRecordToBinaryFile(record, RRN);

updateHeader(getNumberOfRecords()+1);

return RRN;

}

void ProductManager::deleteRecordAtOffset(int offset) {

//--------------------------------------------------------------------------------//

//Delete the record at the offset passed. A record is deleted by reading it from

//the file, changing the primary key to -1, and writing the modified record to the

//original position.

//Preconditions: offset to delete passed as parameter.

//Postconditions: record is read from offset, -1 is placed in its key field, and it

//is written to the file.

//--------------------------------------------------------------------------------//

binaryRecordFile.seekg(offset \* sizeof(DataRecord));

DataRecord deleteRecord;

binaryRecordFile.read(reinterpret\_cast< char \* >(&deleteRecord), sizeof(DataRecord));

binaryRecordFile.seekp(offset \* sizeof(DataRecord));

deleteRecord.setKey(-1);

binaryRecordFile.write(reinterpret\_cast< char \* >(&deleteRecord), sizeof(DataRecord));

}

#pragma mark - Search

DataRecord ProductManager::searchFileForCost(double cost) {

//--------------------------------------------------------------------------------//

//Search the data file for the cost parameter passed.

//Preconditions: cost to search for passed as a parameter.

//Postconditions: the record with a cost matching the parameter passed is returned

//to the caller. If it is not found the key is set to -1 in the returned record.

//--------------------------------------------------------------------------------//

binaryRecordFile.seekg(0);

bool found = false;

DataRecord header;

int numberOfRecords = getNumberOfRecords();

DataRecord record;

for (int i = 0; i < numberOfRecords && !found; i++) {

//std::cout << "READING: ";

binaryRecordFile.read(reinterpret\_cast< char \* >(&record), sizeof(DataRecord));

if (record.getCost() == cost) {

found = true;

}

}

if (!found) {

record.setKey(-1);

}

return record;

}

DataRecord ProductManager::searchFileForDescription(std::string description) {

//--------------------------------------------------------------------------------//

//Search the data file for the description parameter passed.

//Preconditions: description to search for passed as a parameter.

//Postconditions: the record with a description matching the parameter passed is

//returned to the caller. If it is not found the key is set to -1 in the returned

//record.

//--------------------------------------------------------------------------------//

binaryRecordFile.seekg(0);

bool found = false;

DataRecord header;

int numberOfRecords = getNumberOfRecords();

DataRecord record;

for (int i = 0; i < numberOfRecords && !found; i++) {

//std::cout << "READING: ";

binaryRecordFile.read(reinterpret\_cast< char \* >(&record), sizeof(DataRecord));

//std::cout << "COMPARING: " << description << " & " << record.getName() << std::endl;

if (description.compare(record.getName()) == 0) {

//std::cout << "MATCH" << std::endl;

found = true;

}

}

if (!found) {

record.setKey(-1);

}

return record;

}

#pragma mark - Conversion and Split Methods

DataRecord ProductManager::getDataRecordForString(std::string record) {

//--------------------------------------------------------------------------------//

//Convert the string passed into a data record. Each field is checked during

//conversion to see if it is valid, if it is not a message is displayed to the

//console and -1 is set as the key field of the data record. If the record is valid,

///the new record is returned to the caller.

//Preconditions: record to convert passed as a parameter.

//Postconditions: The DataRecord represented by the string passed is created and

//returned or the errors that occurred are displayed to the user.

//--------------------------------------------------------------------------------//

//Get the record elements split by space.

std::vector<std::string> splitRecord;

splitRecord = split(record, ' ', splitRecord);

//splitRecord[0] = key

//splitRecord[1] = name

//splitRecord[2] = code

//splitRecord[3] = cost

int key = convertStringToInt(splitRecord.at(0));

double cost = convertStringToDouble(splitRecord.at(3));

const char \*name = splitRecord.at(1).c\_str();

int code = convertStringToInt(splitRecord.at(2));

if (key == -1) {

//Invalid key passed.

std::cout << "Invalid Primary Key. Must be integer.\n";

} if(splitRecord.at(1).size() > 8) {

//Invalid description passed.

std::cout << "Invalid Description. Can be up to 8 characters.\n";

key = -1;

name = "INVALID";

} if (cost < 0) {

std::cout << "Invalid Cost. Must be double value.\n";

key = -1;

cost = -1;

} if (code < 0) {

std::cout << "Invalid Code. Must be integer value.\n";

key = -1;

code = -1;

}

DataRecord newRecord(key, name, code, cost);

return newRecord;

}

int ProductManager::convertStringToInt(std::string intString) {

//--------------------------------------------------------------------------------//

//Convert the string passed into an integer and return the integer value.

//Preconditions: string representation of an integer passed as parameter.

//Postconditions: the string passed is parsed and the integer value of the string

//is returned to the caller.

//--------------------------------------------------------------------------------//

int convertedInteger;

std::istringstream converter(intString);

if (converter >> convertedInteger) {

return convertedInteger;

} else {

return -1;

}

}

double ProductManager::convertStringToDouble(std::string doubleString) {

//--------------------------------------------------------------------------------//

//Convert the string passed into a double and return the double value.

//Preconditions: string representation of a double passed as parameter.

//Postconditions: the string passed is parsed and the double value of the string

//is returned to the caller.

//--------------------------------------------------------------------------------//

double convertedDouble;

std::istringstream converter(doubleString);

if (converter >> convertedDouble) {

return convertedDouble;

} else {

return -1;

}

return convertedDouble;

}

std::vector<std::string> ProductManager::split(const std::string &s, char delim, std::vector<std::string> &elems) {

//--------------------------------------------------------------------------

//Split a string using the delimeter specified. The vector that will contain

//the split string is passed by reference

//Precondition: a string to split, delimeter to split by, and vector to store

//the split strings in passed as parameters.

//Postcondition: The method splits the string by the parameter and stores it

//in the elems vector. This vector is also returned to the caller.

//--------------------------------------------------------------------------

std::stringstream ss(s);

std::string item;

while (getline(ss, item, delim)) {

elems.push\_back(item);

}

return elems;

}

IndexListManager.h

/\*

---------------------------------------------------------------------------------

CLASS NAME: IndexListManager

PROGRAMMER: Samuel Jentsch

CLASS: CSC 331.001, Fall 2014

INSTRUCTOR: Dr. R. Strader

REFERENCES: C++ How to Program

Paul Deitel & Harvey Deitel

Dr. Strader: assignment information sheet

CLASS PURPOSE:

IndexListManager is responsible for all actions associated with the primary and

secondary index lists, including:

- Creating files for the primary and secondary index lists

- Saving the primary and secondary index lists to files

- Maintaining headers and file integrity for both files.

- Inserting new record information into the lists

- Deleting records from the lists.

- Searching the lists for a specific key or code.

VARIABLE DICTIONARY:

&primaryIndexFile - fstream, reference to the file to read and write the primary

index list to/from.

&secondaryIndexFile - fstream, reference to the file to read and write the

secondary index list to/from.

keyList - vector<IndexRecord>, a vector containing IndexRecords, used to

represent the primary key list.

invertedList - vector<SecondaryIndexRecord>, a vector containing

SecondaryIndexRecords, used to represent the secondary key list.

---------------------------------------------------------------------------------

\*/

#ifndef \_\_TermProject\_Alpha\_\_IndexListManager\_\_

#define \_\_TermProject\_Alpha\_\_IndexListManager\_\_

#include <stdio.h>

#include <sstream>

#include <string>

#include <vector>

#include <iostream>

#include <fstream>

#include "IndexRecord.h"

#include "DataRecord.h"

#include "SecondaryIndexRecord.h"

class IndexListManager {

private:

std::fstream &primaryIndexFile;

std::fstream &secondaryIndexFile;

std::vector<IndexRecord> keyList;

std::vector<SecondaryIndexRecord> invertedList;

template<typename T>

int binarySearch(int key, std::vector<T> searchList);

public:

IndexListManager(std::fstream &primaryIndexFile, std::fstream &secondaryIndexFile);

//\*\*\*Sort\*\*\*//

void sortKeyList();

void sortInvertedList();

//\*\*\*Initalize from new data file

void populateKeyListFromDataRecordFile(std::fstream &dataFile);

//\*\*\*Initialize from existing files\*\*\*//

void populateSecondaryKeyListFromFile();

void populateKeyListFromIndexRecordFile();

//\*\*\*Save to files\*\*\*//

void savePrimaryKeyListToFile();

void saveSecondaryKeyListToFile();

//\*\*\*Display\*\*\*//

void printKeyList();

//\*\*\*Search\*\*\*//

int getRecordOffsetForKey(int key);

std::vector<int> getPrimaryKeysWithCode(int code);

//\*\*\*Add\*\*\*//

void addDataRecordToKeyList(DataRecord newRecord, int RRN);

void addDataRecordToInvertedList(DataRecord record);

//\*\*\*Delete\*\*\*//

void deleteIndexRecordWithKey(int key);

int deleteSecondaryIndexWithKey(int key);

void removePrimaryIndexFromSecondaryKeyWithCode(int code, int primaryKey);

};

#endif /\* defined(\_\_TermProject\_Alpha\_\_IndexListManager\_\_) \*/

IndexListManager.cpp

#include "IndexListManager.h"

#pragma mark - Constructor

IndexListManager::IndexListManager(std::fstream &primaryIndexFile, std::fstream &secondaryIndexFile)

:primaryIndexFile(primaryIndexFile),

secondaryIndexFile(secondaryIndexFile)

{

}

#pragma mark - Populate Lists from New Data File

void IndexListManager::populateKeyListFromDataRecordFile(std::fstream &dataFile) {

//--------------------------------------------------------------------------------//

//BRINGS KEYLIST INTO MEMORY.

//Read strings from the file until it is empty. Convert these strings into records

//for processing.

//Preconditions: indexFile class attribute initialized through constructor.

//Postconditions: records are created and added to the keyList for each string

//in the file.

//--------------------------------------------------------------------------------//

std:: cout << "POPULATE\n";

dataFile.seekg(0);

int lines = 1;

DataRecord header;

dataFile.read(reinterpret\_cast<char\*>(&header), sizeof(DataRecord));

//std::cout << "HEADER: " << header.getKey() << std::endl;

DataRecord record;

IndexRecord indexRecord;

for (int i = 0; i < header.getKey(); i++) {

dataFile.read(reinterpret\_cast<char\*>(&record), sizeof(DataRecord));

//std::cout << "GOT RECORD: ";

//record.printRecord();

addDataRecordToKeyList(record, lines);

addDataRecordToInvertedList(record);

lines++;

}

}

#pragma mark - Populate Lists from Existing Files

void IndexListManager::populateKeyListFromIndexRecordFile() {

//--------------------------------------------------------------------------------//

//BRINGS KEYLIST INTO MEMORY.

//Read strings from the file until it is empty. Convert these strings into records

//for processing.

//Preconditions: indexFile class attribute initialized through constructor.

//Postconditions: records are created and added to the keyList for each string

//in the file.

//--------------------------------------------------------------------------------//

primaryIndexFile.seekg(0);

IndexRecord header;

primaryIndexFile.read(reinterpret\_cast<char\*>(&header), sizeof(IndexRecord));

IndexRecord indexRecord;

for (int i = 0; i < header.getKey(); i++) {

primaryIndexFile.read(reinterpret\_cast<char\*>(&indexRecord), sizeof(IndexRecord));

keyList.push\_back(indexRecord);

}

}

void IndexListManager::populateSecondaryKeyListFromFile() {

//--------------------------------------------------------------------------------//

//Populate secondary key list from file.

//Preconditions: none

//Postconditions: Key list is populated from the file.

//--------------------------------------------------------------------------------//

secondaryIndexFile.seekg(0);

SecondaryIndexRecord headerRecord;

secondaryIndexFile.read(reinterpret\_cast<char\*>(&headerRecord), sizeof(SecondaryIndexRecord));

int listSize;

int code;

int primaryKey;

for (int i = 0; i < headerRecord.getKey(); i++) {

secondaryIndexFile.read(reinterpret\_cast<char\*>(&code), sizeof(int));

secondaryIndexFile.read(reinterpret\_cast<char\*>(&listSize), sizeof(int));

SecondaryIndexRecord record(code);

for (int j = 0; j < listSize; j++) {

secondaryIndexFile.read(reinterpret\_cast<char\*>(&primaryKey), sizeof(int));

record.addPrimaryKey(primaryKey);

}

invertedList.push\_back(record);

}

sortInvertedList();

}

#pragma mark - Save Lists to Files

void IndexListManager::savePrimaryKeyListToFile() {

//--------------------------------------------------------------------------------//

//Save key list to file.

//Preconditions: none

//Postconditions: key list saved to file.

//--------------------------------------------------------------------------------//

primaryIndexFile.seekp(0);

IndexRecord header((int)keyList.size(), 0);

primaryIndexFile.write(reinterpret\_cast<char\*>(&header), sizeof(IndexRecord));

//std::cout << "KEYLIST LENGTH: " << keyList.size() << std::endl;

for (int i = 0; i < keyList.size(); i++) {

//std::cout << "ADDING " << keyList.at(i).getKey() << " " << keyList.at(i).getRelativeOffset() << "\n";

primaryIndexFile.write(reinterpret\_cast<char\*>(&keyList.at(i)), sizeof(IndexRecord));

//std::cout << "SUCCESS\n";

}

}

void IndexListManager::saveSecondaryKeyListToFile() {

//--------------------------------------------------------------------------------//

//Save key list to file.

//Preconditions: none

//Postconditions: key list saved to file.

//--------------------------------------------------------------------------------//

secondaryIndexFile.seekp(0);

SecondaryIndexRecord headerRecord((int) invertedList.size());

secondaryIndexFile.write(reinterpret\_cast<char\*>(&headerRecord), sizeof(SecondaryIndexRecord));

for (int i = 0; i < (int)invertedList.size(); i++) {

//Write out the code

int code = invertedList.at(i).getKey();

secondaryIndexFile.write(reinterpret\_cast<char\*>(&code), sizeof(int));

//Now write out the number of primary keys following

int listSize = (int)invertedList.at(i).primaryKeys.size();

secondaryIndexFile.write(reinterpret\_cast<char\*>(&listSize), sizeof(int));

//Now write out the primary keys

for (int j = 0; j < (int)invertedList.at(i).primaryKeys.size(); j++) {

int primaryKey = invertedList.at(i).primaryKeys.at(j);

secondaryIndexFile.write(reinterpret\_cast<char\*>(&primaryKey), sizeof(int));

}

}

}

#pragma mark - Sort

void IndexListManager::sortKeyList() {

//Such efficiency.

//Much STL library usage.

//Wow.

std::sort(keyList.begin(), keyList.end());

}

void IndexListManager::sortInvertedList() {

//Such efficiency.

//Much STL library usage.

//Wow.

std::sort(invertedList.begin(), invertedList.end());

}

#pragma mark - Display

void IndexListManager::printKeyList() {

//--------------------------------------------------------------------------------//

//Display the key list to the console

//Preconditions: none

//Postconditions: key list is displayed to console.

//--------------------------------------------------------------------------------//

std::cout << "KEY LIST\n";

for (int i = 0; i < keyList.size(); i++) {

std::cout << keyList.at(i).getKey() << " " << keyList.at(i).getRelativeOffset() << "\n";

}

std::cout << "SECONDARY KEY LIST\n";

for (int i = 0; i < invertedList.size(); i++) {

std::cout << invertedList.at(i).getKey() << " ";

invertedList.at(i).printPrimaryKeys();

std::cout << std::endl;

}

}

#pragma mark - Search

int IndexListManager::getRecordOffsetForKey(int key) {

//--------------------------------------------------------------------------------//

//Read the record from the Data Record File that has a key matching the key passed.

//Call binary search of the Key List to find the RRN for the record with the

//matching key. Intialize a Data Record from the file and return to the caller.

//Preconditions: key passed as integer to search for, initialize, and return.

//Postconditions: A Data Record is initialized from the file (using the RRN from the

//key list) and returned to the caller.

//--------------------------------------------------------------------------------//

int indexForRecord = -1;

int indexRecordIndex = binarySearch(key, keyList);

if (indexRecordIndex != -1) {

indexForRecord = keyList.at(indexRecordIndex).getRelativeOffset();

}

return indexForRecord;

}

std::vector<int> IndexListManager::getPrimaryKeysWithCode(int code) {

//--------------------------------------------------------------------------------//

//Perform a binary search on the secondary list for the key passed as parameter

//Return the primary keys associated with the code passed.

//Preconditions: key passed as integer to search for.

//Postconditions: primary keys associated with the code passed are returned to the

//caller, if the code was not found, an empty vector is returned.

//--------------------------------------------------------------------------------//

int listIndex = binarySearch(code, invertedList);

std::vector<int> keys;

if (listIndex != -1) {

keys = invertedList.at(listIndex).primaryKeys;

}

return keys;

}

template<typename T>

int IndexListManager::binarySearch(int key, std::vector<T> searchList) {

//--------------------------------------------------------------------------------//

//Perform a binary search on the primary key list for the int key passed as parameter

//Return the index of the record in the key list with a key matching the key passed.

//Preconditions: key passed as integer to search for.

//Postconditions: Index of record with key passed is returned to caller. -1 is

//returned if the key was not found.

//--------------------------------------------------------------------------------//

int index = -1;

int top\_bound = (int)searchList.size() - 1;

int bottom\_bound = 0;

int mid;

while (top\_bound >= bottom\_bound) {

mid = (top\_bound+bottom\_bound)/2;

if (searchList.at(mid).getKey() == key) {

return mid;

} else if(searchList.at(mid).getKey() < key) {

bottom\_bound = mid + 1;

} else {

top\_bound = mid - 1;

}

}

return index;

}

#pragma mark - Add

void IndexListManager::addDataRecordToKeyList(DataRecord newRecord, int RRN) {

//--------------------------------------------------------------------------------//

//Add the data record passed to the primary key list. Set the RRN to the parameter

//passed.

//Preconditions: record to add and RRN for record passed as parameters.

//Postconditions: Index record is created for the data record passed and added to

//the key list.

//--------------------------------------------------------------------------------//

IndexRecord indexRecord;

indexRecord.setKey(newRecord.getKey());

indexRecord.setRelativeOffset(RRN);

keyList.push\_back(indexRecord);

sortKeyList();

}

void IndexListManager::addDataRecordToInvertedList(DataRecord record) {

//--------------------------------------------------------------------------------//

//Add the data record passed to the secondary key list. If an entry for the records

//code already exists, add the record's primary key to the secondary index's entry.

//Otherwise create a new secondary index and add it to the inverted list.

//Preconditions: record to add passed as parameter.

//Postconditions: the records key is added to an existing entry or a new

//SecondaryIndex is instantiated and added to the inverted list.

//--------------------------------------------------------------------------------//

int code = record.getCode();

int index = binarySearch(code, invertedList);

if (index == -1) {

//If search for code in existing list returns -1 add code to the list

SecondaryIndexRecord newSecondaryIndex(code);

newSecondaryIndex.addPrimaryKey(record.getKey());

invertedList.push\_back(newSecondaryIndex);

sortInvertedList();

} else {

//Otherwise, add the primary key to the existing entry

invertedList.at(index).addPrimaryKey(record.getKey());

}

}

# pragma mark - Delete

void IndexListManager::deleteIndexRecordWithKey(int key) {

//--------------------------------------------------------------------------------//

//Delete the index record from the primary key list with a key matching the

//parameter passed.

//Preconditions: key to delete passed as parameter.

//Postconditions: If a record with a key matching the key passed is found, it is

//removed from the primary key list.

//--------------------------------------------------------------------------------//

int listIndex = binarySearch(key, keyList);

std::vector<IndexRecord>::iterator iterator = keyList.begin();

keyList.erase(iterator + listIndex);

}

int IndexListManager::deleteSecondaryIndexWithKey(int key) {

//--------------------------------------------------------------------------------//

//Delete the secondary index record from the secondary key list with a key matching

//the parameter passed.

//Preconditions: key to delete passed as parameter.

//Postconditions: If a record with a key matching the key passed is found, it is

//removed from the secondary key list.

//--------------------------------------------------------------------------------//

int listIndex = binarySearch(key, invertedList);

//std::cout << "DELETING AT INDEX: " << listIndex << std::endl;

if (listIndex != -1) {

std::vector<SecondaryIndexRecord>::iterator iterator = invertedList.begin();

invertedList.erase(iterator + listIndex);

}

return listIndex;

}

void IndexListManager::removePrimaryIndexFromSecondaryKeyWithCode(int code, int primaryKey) {

//--------------------------------------------------------------------------------//

//When a primary key is deleted, it must be removed from the secondary key list.

//This method deletes the primary key from the secondary key list, removing the key

//and leaving the entry if there are more keys associated with the code, or removing

//the entry entirely if deleting the key removed the last item associated with the

//code.

//Preconditions: code to delete from and key to delete passed as parameter.

//Postconditions: key is removed from secondary index entry. Entry is removed from

//inverted list if empty.

//--------------------------------------------------------------------------------//

int listIndex = binarySearch(code, invertedList);

//std::cout << "DELETING AT INDEX: " << listIndex << std::endl;

SecondaryIndexRecord \*deleteRecord = &invertedList.at(listIndex);

deleteRecord->removePrimaryKey(primaryKey);

if (deleteRecord->primaryKeys.size() == 0) {

deleteSecondaryIndexWithKey(deleteRecord->getKey());

}

}

DataRecord.h

/\*

-----------------------------------------------------------------------

CLASS NAME: DataRecord

PROGRAMMER: Samuel Jentsch

CLASS: CSC 331.001, Fall 2014

INSTRUCTOR: Dr. R. Strader

REFERENCES: C++ How to Program

Paul Deitel & Harvey Deitel

Dr. Strader: assignment information sheet

CLASS PURPOSE:

a. This class is used to hold each of the attributes for a data record.

Records are in the form:

INT CHAR[8] INT DOUBlE

KEY DESCRIPTION COUNT NEXTPART

VARIABLE DICTIONARY:

key - int, contains the primary key for the record.

name - char[8], an 8 character text name for the record.

code - int, the code associated with the record.

cost - double, the cost associated with the record.

-----------------------------------------------------------------------

\*/

#ifndef \_\_\_31\_Project4\_\_DataRecord\_\_

#define \_\_\_31\_Project4\_\_DataRecord\_\_

#include <stdio.h>

#include <iostream>

class DataRecord {

private:

int key;

char name[8];

int code;

double cost;

public:

DataRecord();

DataRecord(int key, const char \*name, int code, double cost);

//\*\*\*key\*\*\*//

void setKey(int key);

int getKey() const;

//\*\*\*name\*\*\*//

void setName(const char \*name);

std::string getName();

//\*\*\*code\*\*\*//

void setCode(int code);

int getCode() const;

//\*\*\*cost\*\*\*//

void setCost(double cost);

double getCost() const;

void printRecord();

};

#endif /\* defined(\_\_\_31\_Project4\_\_DataRecord\_\_) \*/

DataRecord.cpp

#include "DataRecord.h"

#pragma mark - Constructors

DataRecord::DataRecord() {

//Default constructor.

//Initialize data members to default values.

setCost(0);

setCode(0);

setKey(0);

char blankName[8] = {};

setName(blankName);

}

DataRecord::DataRecord(int key, const char\* name, int code, double cost)

:key(key), code(code), cost(cost)

{

char blankName[8] = {};

setName(blankName);

setName(name);

}

#pragma mark - Getters/Setters

void DataRecord::setKey(int key) {

//--------------------------------------------------------------------------------//

//Preconditions: key parameter passed as integer.

//Postconditions: the key attribute is set to the parameter passed.

//--------------------------------------------------------------------------------//

DataRecord::key = key;

}

int DataRecord::getKey() const {

//--------------------------------------------------------------------------------//

//Postconditions: key is returned to the calling method.

//--------------------------------------------------------------------------------//

return key;

}

void DataRecord::setName(const char \*nameString) {

//--------------------------------------------------------------------------------//

//Preconditions: count parameter passed as integer.

//Postconditions: the count attribute is set to the parameter passed.

//--------------------------------------------------------------------------------//

char c = nameString[0];

int count = 0;

while (isalnum(c)) {

name[count] = c;

c = nameString[++count];

}

name[count] = '\0';

}

std::string DataRecord::getName() {

//--------------------------------------------------------------------------------//

//Preconditions: description string pointer passed as string

//Postconditions: a string containing the values of the class attribute description

//is created and returned to the caller.

//--------------------------------------------------------------------------------//

std::string nameString(name);

return nameString;

}

void DataRecord::setCode(int code) {

//--------------------------------------------------------------------------------//

//Preconditions: count parameter passed as integer.

//Postconditions: the count attribute is set to the parameter passed.

//--------------------------------------------------------------------------------//

DataRecord::code = code;

}

int DataRecord::getCode() const {

//--------------------------------------------------------------------------------//

//Postconditions: count is returned to the calling method.

//--------------------------------------------------------------------------------//

return code;

}

void DataRecord::setCost(double cost) {

//--------------------------------------------------------------------------------//

//Preconditions: nextPart parameter passed as integer.

//Postconditions: the nextPart attribute is set to the parameter passed.

//--------------------------------------------------------------------------------//

DataRecord::cost = cost;

}

double DataRecord::getCost() const {

//--------------------------------------------------------------------------------//

//Postconditions: nextPart is returned to the calling method.

//--------------------------------------------------------------------------------//

return cost;

}

#pragma mark - Print

void DataRecord::printRecord() {

//--------------------------------------------------------------------------------//

//Preconditions: none

//Postconditions: the attributes of the DataRecord instance are printed to the

//console.

//--------------------------------------------------------------------------------//

std::cout << key << " " << name << " " << code << " " << cost << std::endl;

}

KeyRecord.h

/\*

-----------------------------------------------------------------------

CLASS NAME: KeyRecord

PROGRAMMER: Samuel Jentsch

CLASS: CSC 331.001, Fall 2014

INSTRUCTOR: Dr. R. Strader

REFERENCES: C++ How to Program

Paul Deitel & Harvey Deitel

Dr. Strader: assignment information sheet

CLASS PURPOSE:

This class acts as a basic record containing a key. This allows for

subclasses of KeyRecord to be sorted and compared based on their

key attribute.

VARIABLE DICTIONARY:

key - int, contains the key for the record.

-----------------------------------------------------------------------

\*/

#ifndef \_\_TermProject\_Beta\_\_KeyRecord\_\_

#define \_\_TermProject\_Beta\_\_KeyRecord\_\_

#include <stdio.h>

class KeyRecord {

protected:

int key;

public:

//\*\*\*key\*\*\*//

void setKey(int key);

int getKey() const;

//\*\*\*Overloaded Constructors\*\*\*//

bool operator< (const KeyRecord &rec) const;

bool operator> (const KeyRecord &rec) const;

bool operator== (const KeyRecord &rec) const;

};

#endif /\* defined(\_\_TermProject\_Beta\_\_KeyRecord\_\_) \*/

KeyRecord.cpp

#include "KeyRecord.h"

#pragma mark - Getters/Setters

void KeyRecord::setKey(int key) {

//--------------------------------------------------------------------------------//

//Preconditions: key parameter passed as integer.

//Postconditions: the key attribute is set to the parameter passed.

//--------------------------------------------------------------------------------//

KeyRecord::key = key;

}

int KeyRecord::getKey() const {

//--------------------------------------------------------------------------------//

//Postconditions: key is returned to the calling method.

//--------------------------------------------------------------------------------//

return key;

}

#pragma mark - Overridden Operators

//Operators are overridden to compare the key class attribute.

bool KeyRecord::operator< (const KeyRecord &rec) const {

return key < rec.getKey();

}

bool KeyRecord::operator> (const KeyRecord &rec) const {

return key > rec.getKey();

}

bool KeyRecord::operator== (const KeyRecord &rec) const {

return key == rec.getKey();

}

IndexRecord.h

/\*

-----------------------------------------------------------------------

CLASS NAME: IndexRecord

PROGRAMMER: Samuel Jentsch

CLASS: CSC 331.001, Fall 2014

INSTRUCTOR: Dr. R. Strader

REFERENCES: C++ How to Program

Paul Deitel & Harvey Deitel

Dr. Strader: assignment information sheet

CLASS PURPOSE:

a. This class is used to hold an index record referencing a record in a

data file.

Records are in the form:

INT INT

KEY RRN

VARIABLE DICTIONARY:

relativeOffset - int, contains the relative offset for the record

referenced by the index record

-----------------------------------------------------------------------

\*/

#ifndef \_\_\_31\_Project5\_\_IndexRecord\_\_

#define \_\_\_31\_Project5\_\_IndexRecord\_\_

#include <stdio.h>

#include "KeyRecord.h"

class IndexRecord: public KeyRecord {

private:

int relativeOffset;

public:

IndexRecord();

IndexRecord(int key, int relativeOffset);

//\*\*\*relativeOffset\*\*\*//

void setRelativeOffset(int relativeOffset);

int getRelativeOffset() const;

};

#endif /\* defined(\_\_\_31\_Project5\_\_IndexRecord\_\_) \*/

IndexRecord.cpp

#include "IndexRecord.h"

#pragma mark - Constructors

IndexRecord::IndexRecord() {

//Default constructor

setKey(0);

setRelativeOffset(0);

}

IndexRecord::IndexRecord(int key, int relativeOffset) {

setKey(key);

setRelativeOffset(relativeOffset);

}

#pragma mark - Getters/Setters

void IndexRecord::setRelativeOffset(int relativeOffset) {

//--------------------------------------------------------------------------------//

//Preconditions: relativeOffset parameter passed as integer.

//Postconditions: the relativeOffset attribute is set to the parameter passed.

//--------------------------------------------------------------------------------//

IndexRecord::relativeOffset = relativeOffset;

}

int IndexRecord::getRelativeOffset() const {

//--------------------------------------------------------------------------------//

//Postconditions: RRN is returned to the calling method.

//--------------------------------------------------------------------------------//

return relativeOffset;

}

SecondaryIndexRecord.h

/\*

-----------------------------------------------------------------------

CLASS NAME: SecondaryIndexRecord

PROGRAMMER: Samuel Jentsch

CLASS: CSC 331.001, Fall 2014

INSTRUCTOR: Dr. R. Strader

REFERENCES: C++ How to Program

Paul Deitel & Harvey Deitel

Dr. Strader: assignment information sheet

CLASS PURPOSE:

Represents a SecondaryIndexRecord. Contains a vector acting as a list

of all the primary keys associated with the records code.

Allows for insertion of additional primary keys, deletion of keys, and

display of primary keys contained int the primaryKeys vector.

VARIABLE DICTIONARY:

primaryKeys - vector<int>, a vector containing all of the primary keys

associated with the records code.

-----------------------------------------------------------------------

\*/

#ifndef \_\_TermProject\_Beta\_\_SecondaryIndexRecord\_\_

#define \_\_TermProject\_Beta\_\_SecondaryIndexRecord\_\_

#include <stdio.h>

#include <vector>

#include <iostream>

#include "KeyRecord.h"

class SecondaryIndexRecord: public KeyRecord {

private:

int binarySearch(int key, std::vector<int> searchList);

public:

std::vector<int> primaryKeys;

SecondaryIndexRecord();

SecondaryIndexRecord(int key);

void addPrimaryKey(int primaryKey);

void removePrimaryKey(int primaryKey);

void printPrimaryKeys();

};

#endif /\* defined(\_\_TermProject\_Beta\_\_SecondaryIndexRecord\_\_) \*/

SecondaryIndexRecord.cpp

#include "IndexRecord.h"

#include "SecondaryIndexRecord.h"

SecondaryIndexRecord::SecondaryIndexRecord() {

}

SecondaryIndexRecord::SecondaryIndexRecord(int key) {

setKey(key);

}

void SecondaryIndexRecord::addPrimaryKey(int primaryKey) {

//--------------------------------------------------------------------------------//

//Add primary key to primary key list.

//Preconditions: primary key to add passed as parameter.

//Postconditions: the primary key is added to the primary key list.

//--------------------------------------------------------------------------------//

primaryKeys.push\_back(primaryKey);

std::sort(primaryKeys.begin(), primaryKeys.end());

}

void SecondaryIndexRecord::printPrimaryKeys() {

//--------------------------------------------------------------------------------//

//Print the list of primary keys.

//Preconditions: none.

//Postconditions: primary keys displayed to console.

//--------------------------------------------------------------------------------//

for (int i = 0; i < primaryKeys.size(); i++) {

std::cout << primaryKeys.at(i) << " ";

}

}

#pragma mark - Delete

void SecondaryIndexRecord::removePrimaryKey(int primaryKey) {

//--------------------------------------------------------------------------------//

//Delete a primary key from the list associated with the secondary index.

//Preconditions: primary key to remove passed as parameter.

//Postconditions: the primary key is removed from the primary key list.

//--------------------------------------------------------------------------------//

int listIndex = binarySearch(primaryKey, primaryKeys);

std::vector<int>::iterator iterator = primaryKeys.begin();

//std::cout << "REMOVING PRIMARY KEY AT INDEX: " << listIndex << std::endl;

primaryKeys.erase(iterator + listIndex);

}

int SecondaryIndexRecord::binarySearch(int key, std::vector<int> searchList) {

//--------------------------------------------------------------------------------//

//Perform a binary search on the primary key list for the int key passed as parameter

//Return the index of the record in the key list with a key matching the key passed.

//Preconditions: key passed as integer to search for.

//Postconditions: Index of record with key passed is returned to caller. -1 is

//returned if the key was not found.

//--------------------------------------------------------------------------------//

int index = -1;

int top\_bound = (int)searchList.size() - 1;

int bottom\_bound = 0;

int mid;

while (top\_bound >= bottom\_bound) {

mid = (top\_bound+bottom\_bound)/2;

if (searchList.at(mid) == key) {

return mid;

} else if(searchList.at(mid) < key) {

bottom\_bound = mid + 1;

} else {

top\_bound = mid - 1;

}

}

return index;

}