25 points possible Due 3/24/2020

Programming Assignment 3

Statement of Work

1 **Overview**

You have been contract by a wholesale computer equipment supplier. The supplier sells computers, computer parts, and computer peripherals to a large number of retail stores throughout the United States. The company would like to create a database of all its sales representatives. This database must be capable of maintaining the employee ID, employee name, department, and annual salary of each sales representative. In addition it must also maintain a list of stores and their locations for each employee. The third phase of development will be to create the database of employee sales representatives.

2 **Requirements**

The student shall define, develop, document, prototype, test, and modify as required the software system.

2.1 This software system shall consist of four source files (.cpp) and four header files (.h) defining four classes. The first class, which was developed in phase one and modified in phase two to hold a list of stores, is the **EmployeeRecord** class. It will be modified so that it can be used to build a binary tree of employee records. The second class shall be the Store class provided in phase 2. The third class shall be the **CustomerList** class developed in phase 2. The fourth class, which shall be called **EmployeeDatabase**, shall define and maintain a binary tree of EmployeeRecord objects.

2.2 The class **EmployeeRecord**, as used in Phase 2 shall be modified so that it can be used to build a binary tree of EmployeeRecord objects.

2.2.1 Two public pointers to a class of type **EmployeeRecord** called **m_pLeft** and **m_pRight** shall be added to the EmployeeRecord class.

2.2.2 The public function **void removeCustomerList()** will be added to the EmployeeRecord class. This function sets the pointer to a CustomerList object to NULL. This function is used by the removeEmployee() function in the EmployeeDatabase class. See details of how this function is used in the hints section below.

2.2.3 The public function **void destroyCustomerList()** will be added to the EmployeeRecord class. This function deletes the CustomerList object and sets the pointer to a CustomerList object to NULL. It is used by the removeEmployee() function in the EmployeeDatabase class. See details of how this function is used in the hints section below. **NOTE: You will have to modify the destructor of EmployeeRecord to check to see if m_pCustomerList is not NULL before calling delete m_pCustomerList.**

2.3 The class **EmployeeDatabase**, shall implement a binary tree of EmployeeRecord objects.

2.3.1 The class **EmployeeDatabase** shall contain private variables called **m_pRoot** which shall be a pointer to an EmployeeRecord object and **inFile** which shall be an ifstream object.

2.3.2 The class **EmployeeDatabase** shall contain the following public functions:

2.3.2.1 **EmployeeDatabase()** and **~EmployeeDatabase** – A constructor and a destructor. The destructor function shall take care of deleting all EmployeeRecord objects in the tree to avoid memory leaks. It will do this by initializing a recursive traversal to the tree calling the private function **destroyTree(m_pRoot)**

2.3.2.2 **bool addEmployee(EmployeeRecord *e)** – This function will take a pointer to a completed EmployeeRecord object and insert the object into the binary tree.

2.3.2.3 **EmployeeRecord *getEmployee(int ID)** – This function will take an employee ID as an argument. It will search the tree and return a pointer to the employee whose ID matches the function argument. It will return NULL if it fails to find the employee in the tree.

2.3.2.4 **EmployeeRecord *removeEmployee(int ID)** – This function will take an employee ID as an argument. It will search the tree, locate (if it exists) the employee record, remove it from the tree and return it. It will return NULL if it failed to find the employee. Note, care must be taken when the EmployeeRecord removed has two children. See the notes below on how to handle this occurance.

2.3.2.5 **void printEmployeeRecords()** – This function shall call the private printEmployeeRecords() function passing in the root of the tree. This shall initialize a recursive traversal to print all records in the tree.

2.3.2.6 **bool buildDatabase(char *dataFile)** – This function shall take a char array specifying the name of the data file. It will read and build the database. This function is provided by the instructor. See below.

2.3.3 The class **EmployeeDatabase** shall contain the following private functions:

2.3.3.1 **void printEmployeeRecords(EmployeeRecord *rt)** – This function overloads the public printEmployeeRecords() function and shall perform an in-order recursive traversal of the entire tree and print all data on all employees in the database. (Note: requirements for the printRecord() function in the EmployeeRecord class are the same as in phase 2.)

2.3.3.2 **void destroyTree(EmployeeRecord *rt) –** This function shall recursively traverse the entire tree and delete all nodes in the tree.

3 Deliverables

These products shall be delivered to the instructor **electronically via Canvas** not later than **Tuesday, March 24, 2020**.

3.1 Sprint Report – The student shall provide a filled out Sprint Report form.

3.2 Program Source Files – The student shall provide fully tested electronic copies of the .cpp and .h files.

What is the purpose of the removeCustomerList() and destroyCustomerList() functions added to EmployeeRecord?

The function **removeCustomerList()** is used to set the m_pCustomerList pointer to null. This function does **NOT** delete the object that m_pCustomerList points to. Normally this would result in a memory leak. See the explanation below on how the delete function in a binary tree works to see why we have to do this in one particular case.

The function **destroyCustomerList()** is used to delete the CustomerList object that an EmployeeRecord's m_pCustomerList is pointing to then set the pointer to null. See the explanation below on how the delete function in a binary tree works to see why we have to do this in one particular case.

Why is there a problem with deleting a node with two children in the EmployeeDatabase binary tree?

In the delete algorithm for a binary tree one of the cases which must be handled is when the node being removed from the tree has two children. As described in the delete algorithm on the web pages of binary tree algorithms the procedure is to copy all information from a leaf node into the node to be deleted then remove the leaf from the tree. In this situation however we have the problem of the CustomerList which maintains a list of Stores external to the EmployeeRecord. If you copy all data from one EmployeeRecord object into another you also copy the CustomerList pointer and thus have two EmployeeRecords each with a pointer to the same CustomerList object.

When you delete the node that the data was copied from its' destructor will destroy the CustomerList (the same one that both nodes have a pointer to) leaving the second EmployeeRecord with a dangling pointer. To prevent this you should modify the delete algorithm as follows when deleting a node with two children:

1. Create a new EmployeeRecord. This will also create its' own instance of CustomerList which is not needed. So after creating it call destroyCustomerList on this new instance of EmployeeRecord.
2. Copy all data from the node to be deleted into this new EmployeeRecord. This will also copy the pointer to the CustomerList object. Just briefly you will have two instances of EmployeeRecord both with pointers to the same instance of CustomerList.
3. Search the left sub-tree of the node to be "removed" to find the node with the largest key. This one will be used to overwrite the node to be "removed."
4. Copy all data from the node in the left sub-tree with the largest key into the node to be "removed". This will also copy the pointer to its' CustomerList leaving both nodes with a pointer to the same CustomerList object.
5. Call removeCustomerList() on the EmployeeRecord that you copied FROM. It will set its' m_pCustomerList pointer to null, but will not delete the CustomerList object. This will leave that instance of CustomerList with the node you are overwriting.
6. Remove from the tree the node you just copied FROM and delete it. Since it's pointer to CustomerList is now NULL it will not try to delete its' CustomerList.
7. Return the new EmployeeRecord that is now a duplicate of the one "removed" from the tree.

25 points possible Due 3/24/2020

The following information will be helpful in this assignment:

In the download zip file there is a text file named **Prog3Data.txt** which contains all the data needed to build an employee data base of 10 employees each with four stores in the CustomerList. To read this data see the code snippets below. Comments at the beginning of the data file explain the format of the data found in it.

Include the following function in your EmployeeDatabase class (this function is also included in the zip file so you don't have to copy and paste it out of this document):

```
//-------------------------------------------------------
// Build the database
//-------------------------------------------------------
bool EmployeeDatabase::buildDatabase(char *dataFile)
{
     bool OK = true;
     int numEmp, id, dept, numStores, sID;
     double sal;
     EmployeeRecord *empRec;
     CustomerList *theList;
     Store *theStore;
     char inStr[128];
     char fName[32];
     char lName[32];
     char sName[64];
     char sAddr[64];
     char sSt[32];
     char sCity[32];
     char sZip[12];


     inFile.open(dataFile, ifstream::in);
    if(!inFile.is_open())
    {
        // inFile.is_open() returns false if the file could not be
found or
        //    if for some other reason the open failed.
        cout << "Unable to open file" << dataFile << "\nProgram
terminating...\n";
          cout << "Press Enter to continue...";
          getc(stdin);
        return false;
    }

     // Get number of employees
```

```
     getNextLine(inStr, 128);
     numEmp = atoi(inStr);
     for(int i=0; i<numEmp; i++)
     {
          // Instantiate an EmployeeRecord
          empRec = new EmployeeRecord();
          // Read and set the ID
          getNextLine(inStr, 127);
          id = atoi(inStr);
          empRec->setID(id);
          // Read and set the name
          getNextLine(fName, 31);
          getNextLine(lName, 31);
          empRec->setName(fName, lName);
          // Read and set the Department ID
          getNextLine(inStr, 127);
          dept = atoi(inStr);
          empRec->setDept(dept);
          // Read and set the Salary
          getNextLine(inStr, 127);
          sal = atof(inStr);
          empRec->setSalary(sal);
          // Get the customer list
          theList = empRec->getCustomerList();
          // Get the number of stores
          getNextLine(inStr, 127);
          numStores = atoi(inStr);
          for(int j=0; j<numStores; j++)
          {
               // Read the store ID
               getNextLine(inStr, 127);
               sID = atoi(inStr);
               // Read the store name
               getNextLine(sName, 63);
               // Read the store address
               getNextLine(sAddr, 63);
               // Read the store city
               getNextLine(sCity, 31);
               // Read the store state
               getNextLine(sSt, 31);
               // Read the store zip
               getNextLine(sZip, 11);
               // Create a new Store object
               theStore = new Store(sID, sName, sAddr, sCity, sSt,
sZip);
```

```
                theList->addStore(theStore);
            }
            cout.flush();
            addEmployee(empRec);
        }
        inFile.close();
        return true;   // Successfully built the database
}
```

Include the following function in your EmployeeDatabase class (this function is also included in the zip file so you don't have to copy and paste it out of this document). Each time it is called it returns a data line from the file. It ignores any comment lines (those starting with a # sign) and any lines that are empty. Check the returned bool value to be sure you have valid data in the char array. This is how the function signals that it has reached the end of a file.

```
//-------------------------------------------
// GetNextLine()
// Read a line from a data file.
// Author: Rick Coleman
// Used by permission
//-------------------------------------------
bool EmployeeDatabase::getNextLine(char *line, int lineLen)
{
    int    done = false;
    while(!done)
    {
        inFile.getline(line, 128);

        if(inFile.good())    // If a line was successfully read
        {
            if(strlen(line) == 0)  // Skip any blank lines
                continue;
            else if(line[0] == '#')  // Skip any comment lines
                continue;
            else return true;    // Got a valid data line so return
with this line
        }
        else
        {
            strcpy(line, "");
            return false;
        }
    } // end while
    return false;
}
```

After opening the data file for reading you can call getNextLine() to get each data line with the command getNextLine(inStr, N-1) where inStr is a char array N characters long. Note: Do not use strlen(inStr) to pass the length of the character array as you want the size of the character array not the length of the string currently stored in it. You use a length of N-1 because you must allow for one character for the NULL terminator.

Use the string conversion functions as in the following examples:

```
     // If the call to getNextLine() is returning string data such as
an employee's
     // first or last name then you can use strcpy() to copy directly
from inStr to
     // another character array. Assume
     // char name[32]; has been declared.
     strcpy(name, inStr);

     // If the call to getNextLine() is returning an integer as a
string, for example "1234"
     // you can convert it to an integer with atoi() which means
"ASCII to Integer".  Assume
     // int ID; has been declared.
     ID = atoi(inStr);

     // If the call to getNextLine() is returning a double as a
string, for example "65536.00"
     // you can convert it to a double with atof() which means "ASCII
to Float".  Assume
     // double sal; has been declared.
     sal = atof(inStr);

     // In the case of adding data to a Store or EmployeeRecord call
the appropriate set function
     // for strings, e.g. st->setStoreName(inStr) and for values st-
>setStoreID(atoi(inStr);
```