**IoT Database Application – Planning and Design**

**Planning**

Specifications:

Create a database management system and calculator for any csv files inputted into the program. The csv files will contain data of water levels, temperature and humidity of nodes. Also make a linear regression calculator which calculates if the regression equation should change based on percent error calculations using real measures of water levels.

Functional Requirements:

The data structures should behave as described in the following.

**Doubly Linked List** - Use a double linked list to hold the daily data files in terms of days. The linked list should hold data of water levels, temperature and humidity of nodes, and to perform calculations for the three parameters. The calculations may be the following: calculating average measure over period of time, calculating maximum measure over a period of time, calculating minimum measure over a period of time, displaying data for the days entered by the user, calculating percent error between a data and an entered value representing the actual measured value. Furthermore, the linear regression calculation will include finding the voltage calibration of a data set of the water levels. (Put the similar operations on the linked list from chapter 3, and allow the user to modify the data files from the application using these operations).

Operations:

1.Inserting a node to the head

2.Inserting a node to the tail

3.Inserting a node (should insert data into data file as well)

4.Deleting a node (should delete data from data file as well)

5.Searching a node

6.Check if empty

7.Is in the linked list

Math Operations:

1.Calculating average measure over period of time

2.Calculating maximum measure over a period of time

3.Calculating minimum measure over a period of time

4.Calculating percent error between a data and an entered value representing the actual measured value

5.Calculating the voltage calibration of a data set of the water levels using linear regression calculation (only available for water level data)

**Queue** – Use a queue to store node upgrade tasks (when there is no priority given to tasks) based on data files of the three types of nodes, the water level measuring, temperature and humidity nodes. (Put the similar operations on the queue from chapter 4, and allow the user to modify the data files from the application using these operations).

Operations:

1.Inserting a node from the tail (should insert data into data file as well)

2.Deleting a node from the head (should delete data from data file as well)

3.Return the front node

4.Check if empty

**Priority Queue** - Use a priority queue to store node upgrade tasks based on data files of the three types of nodes, the water level measuring, temperature and humidity nodes. If any nodes need immediate update. then it should go first, but if the nodes don’t need immediate update, it can go last. (Put the similar operations on the priority queue from chapter 4, and allow the user to modify the data files from the application using these operations).

Operations:

1.Inserting a node anywhere (should insert data into data file as well)

2.Deleting a node from the head (should delete data from data file as well)

3.Return the front node

4.Check if empty

**Stack** – Use a stack to reverse a copy of the linked lists, and display the data from the reverse order for the water levels, temperature and humidity if the user chooses. (Put the similar operations on the stack from chapter 4, and allow the user to modify the data files from the application using these operations).

Operations:

1.Pushing (insert) a node

2.Popping (delete) a node

3.Returning (search) the top node

4.Check if empty

**Binary Search Tree** - Use a binary search tree (from the data files, implement three binary search trees, where the three trees will be for water level data, temperature data and humidity data. For instance, one tree will contain all the water level nodes, and each node contains the most recent water level measurement, and additionally, since this is a binary search tree, the tree will be sorted per the binary search tree criteria). (Put the similar operations on the tree from chapter 6, and allow the user to modify the data files from the application using these operations).

Operations:

1.Inserting a node

2.Deleting a node

3.Searching a node

4.Check if empty

The main function will first load the four data files for water level, temperature, humidity and node upgrade tasks. These data files will be .csv files. The main function will then use these four data files to create the data structures which to operate on and perform calculations for, based on user input.

Non-functional Requirements:

-Quick display of data structures

-Variety of calculations available

-Accurate information displayed

-Applicable to different files with same structures

**Design**

Classes:

dataDLL: This class will contain the doubly linked list (refer to chapter 3 on how to create a doubly linked list and how to implement the operations).

printDLL(): Overload the cout << operator in the functions printDLL() to show the linked list values (see from the directx tutorial how to do this). Use ostringstream to display the values, see pg 414 for more info.

nodeUpgradeQ: This class will contain a queue (refer to chapter 4 on how to create a queue and how to implement the operations).

printQ(): Overload the cout << operator in the functions printQ() to show the queue values (see from the directx tutorial how to do this). Use ostringstream to display the values, see pg 414 for more info.

nodeUpgradePQ: This class will contain a priority queue (refer to chapter 4 on how to create a priority queue and how to implement the operations).

printPQ(): Overload the cout << operator in the functions printPQ() to show the priority queue values (see from the directx tutorial how to do this). Use ostringstream to display the values, see pg 414 for more info.

dataStk: This class will contain a stack (refer to chapter 4 on how to create a stack and how to implement the operations).

printStk(): Overload the cout << operator in the functions printStk() to show the stack values (see from the directx tutorial how to do this). Use ostringstream to display the values, see pg 414 for more info.

dataBST: This class will contain a binary search tree (refer to chapter 4 on how to create a binary search tree and how to implement the operations).

printBST(): Overload the cout << operator in the functions printBST() to show the binary tree values (see from the directx tutorial how to do this). Use ostringstream to display the values, see pg 414 for more info.

DataFiles():

getWaterLevelFile(), getTemperatureFile(), getHumidityFile()

getNodeUpgradesFile()

Files:

Water Level Data File:

Columns:

Date (MM-DD-YYYY), Time in 24 hr format (HH:MM), Water level (decimal value)

Temperature Data File:

Columns:

Date (MM-DD-YYYY), Time in 24 hr format (HH:MM), Temperature (decimal value)

Humidity Data File:

Columns:

Date (MM-DD-YYYY), Time in 24 hr format (HH:MM), Humidity (decimal value)

Node Upgrade Tasks Data File:

Columns:

Priority (whole number value), Task (sentence/string type value)

Main Function:

The application can be started using the git bash terminal. The application will then call the getWaterLevelFile(), getTemperatureFile(), getHumidityFile() functions which are part of the DataFiles class and depending on if there are command line arguments, specific overloaded functions will be called from these three functions. The application can be run regularly with or without any command line arguments. If there are command line arguments, the data files the application will use will be in the default locations as specified by the main function. Otherwise, if the application has the command line arguments specified, it will use these command line arguments (which contain the data files locations) to locate and open the data files. There will be two cases of the user providing the command line arguments or not: in case 1, the user could choose not to provide any command line arguments, and in case 2, the user will provide all four data files (water level, temperature, humidity and node upgrade tasks) location paths to run the application. These data files will be .csv files. Use the member function readFile to read the contents of the provided file path in the command line argument, see pg 407. Control will be shifted to (1).

(2) The control from the getWaterLevelFile(), getTemperatureFile(), getHumidityFile() functions (which are part of the DataFiles class) will resume here. The main function will then call the getNodeUpgradesFile() function. This function will be overloaded, and have two forms where the first form will take no arguments, while the second form will accept the file location path of the node upgrade tasks data file depending on if the user entered any command line arguments. Use a member function readFile to read the contents of the provided file path in the command line argument, see pg 407. Control will then be shifted to (3).

(4) The control from the getNodeUpgradesFile() (which is part of the DataFiles class) will resume here.

The main function will start a infinite while loop which will end only when the input to the cin stream is “close” when asked “[close] or [continue]?”. If “continue” is entered, the loop will continue iterating. This question will first be asked in the while loop. The while loop will essentially function as such:

1.Ask the question “[close] or [continue]?”. If “close” is entered, the loop will stop, and if “continue” is entered, the loop will continue iterating (not actually continuing to the next iteration using the “continue keyword”).

2.Ask the question “Which data would you like to work on [water level], [temperature] [humidity] or [node upgrade tasks]?”. Create a string vector which will store the user’s responses. The first element of this vector will store the data the user would like to work on.

3. Ask the question “Which data structure would you like to work with [doubly linked list], [binary search tree], [queue] or [priority queue]?” (stack is only used by doubly linked list). Add the user’s response to the second element of the vector that will store the data structure the user would like to work with.

4. Ask the question “What operation would you like to perform [insert], [delete], [search] or [math]?” Add the user’s response to the third element of the vector that will store the operation the user would like to perform.

Based on the user’s responses in the three elements of the vector, perform the operations on the data structure with the type of data (water level, temperature, humidity and node upgrade tasks) the user asked to work on. If the specific operation or data structure doesn’t exist for the type of data, then provide a “cerr” error message. Also use a similar member function to the ‘read’ function from ch 7 classes to read input from the user, and to apply the specific operations to the data structures and data files. Use istringstream as well to read the input, see pg 412 for more info.

Once the operation has been finished, the function should proceed to the next iteration to ask the user the previous question again “[close] or [continue]?”.

getWaterLevelFile(), getTemperatureFile(), getHumidityFile() (which is part of the DataFiles class):

**Note:** There will be 6 functions in total, and 2 functions for each get function where both are overloaded functions. If the overloaded function takes no arguments, then use the default data file location path. If the overloaded function takes arguments, then use this argument as the data file location path.

(1) These functions will be overloaded, where one form of the function will accept no arguments, while the other form will accept the command line arguments (containing data file location of either the water level csv file, temperature csv file or humidity csv file) as parameters. After opening these three data files, each of the functions will read the values from the three files and call constructors and/or functions in the dataDLL, dataStk, and dataBST classes to create the doubly linked list, stack (to display a reverse of the double linked list) and binary search tree for each of the three data files.

After the data structures for these three data files are created, the printDLL(), printStk(), printBST() functions in the dataDLL, dataStk, and dataBST classes respectively will be called to print the data of the data structures onto the console. The control will then return to the main function (start control in (2)) without returning anything, or can just use “return” to return from the void functions getWaterLevelFile(), getTemperatureFile(), getHumidityFile() (which is part of the DataFiles class).

getNodeUpgradesFile() (which is part of the DataFiles class):

**Note:** There will be 2 functions in total, and both of these are overloaded functions. If the overloaded function takes no arguments, then use the default data file location path. If the overloaded function takes arguments, then use this argument as the data file location path.

(3) This function will be overloaded, and have two forms where the first form will take no arguments, while the second form will accept the file location path of the node upgrade tasks data file depending on if the user entered any command line arguments. After opening the node upgrade tasks data file, this function will read the values from the node upgrade tasks data file and call constructors and/or functions in the nodeUpgradeQ and nodeUpgradePQ classes to create the queue and priority queue containing the node upgrade tasks.

After the queue and priority queue are created, the printQ() and printPQ() functions in the nodeUpgradeQ and nodeUpgradePQ classes respectively will be called to print the queue and priority queue data structures onto the console. The control will then return to the main function in (4).