Changes: Lab5 Question 5

The struct WindLogType is in a separate file and no longer in IOFile.h. The reason why is because the purpose of a struct is to encapsulate the data. And by putting the struct inside another file it breaks the encapsulation and allows clients to access the struct through a different file. So essentially providing another avenue for uncontrolled access to the data or struct data members.

The CalcMean, and CalcStandardDeviation is poorly designed. It does not allow for reusability. The mean is tied to the speed data member of the windlogType. So, if you want to calculate the mean for another property then you would need to create a separate mean function. I rewrote the calcMean so it is not tied to a data member. In addition, I created a separate calculate total function since the mean should only have a single responsibility. That is to calculate mean and calculating total should be another functions responsibility.

The vector submitted is not minimal and complete. In order to be minimal two functions should not have the same behavior. In my submission there is a function that have the same behavior. Having duplicate methods that do the same thing creates bloat.

The At() function have the same behavior as overloaded [] subscript operator.

So if a client wants to access element at position 1 using At(1) they can use the [1] to access element at position 1 as well. Thus, duplicate functionality.

Secondly, the the vector isn't minimal in that it provides functions that conflict with the abstract behavior of the data structure. The abstract data structure is the human understanding of how the data structure should behave. In the specifications, the vector is defined as an encapsulated dynamic array. So, in the vector I submitted the functions RemoveAt, and PopBack conflict with the abstract behavior of the vector thus is unnecessary. A dynamic array allow does not allow for the removal of an element at a position behavior. Likewise, the PopBack are not abstract behaviors of an encapsulated dynamic array.

Thirdly, the vector is poorly designed because it allows the client to view the capacity and IncreaseMaxCapacity(). These two functions should be hidden from the client. Their purpose is to resize the array when it is full. These two functions are only for maintaining the internal implementation of the data structure. So, removing IncreaseMaxCapacity and GetMaxCapacity improves the program design since the concept of abstraction is followed.

Rationale for the data structures

Role of multimap and map-

Initially, I wanted to load all the data from files to a multimap because the multimap is a red black tree (a balanced tree) as opposed to my non-self-balancing binary search tree. Loading all the data to a non-self-balancing tree would lead to all the data being stored as a linked list. This would mean the binary search tree would no longer be able to perform o(log n) searching, which defeats the whole purpose of using a binary search tree.

Loading all the data into a multimap had its advantages.

Firstly, loading data all the data into a multimap would result in O(log n) insertion time as opposed to binary search tree where O(log n) insertion time isn't guaranteed.

In addition, a multimap allows for duplicate keys and also allows for **guaranteed** O(log n) searching time through the function **equal_range(…)**. The equal_range(..) function is a powerful function that allowed me to search the multimap for a key at a guaranteed O(log n) time. All the required menu options required some sort of retrieval of a data based off a month and year. So, a multimap key created using a class with month and year as its members was appropriate.

Loading all the data into a multimap had its disadvantages as well.

Firstly, there are far superior data structures for fast insertion of data from the files to the structure. For example, a stack and queue allows for inserts at o(1) time guaranteed. But both stack and queue aren't appropriate for this situation since the data structure needs to be frequently searched and accessed fast. A stack and queue has o(n) search and access time which is not suitable.

Secondly, a multimap will not accept duplicate keys. This was problematic! In the data files there were too many records with the same data and time. For example, 1/01/2016 9:10 record appeared in two separate files. So, loading all the data from the files to a multimap with month/year as a key would result in the duplicate records being kept. The only solution I had to this issue was to first load all the data into a map, which would remove the duplicate date and time records. The date and time would be need to be the map key to facilitate this. Once all the data was loaded into the map it would then be inserted into the multimap. The downside to this solution was that extra time was essentially being spent on loading the data.

The decision I made in the end was to load all the data from the files to a map to remove the duplicate records. Then insert the records into a multimap. Ultimately, allowing for guaranteed o(log n) search time for a given month and year using equal_range function was the main reason for my decision.

Role of vector-

For all the required menu options, the records were retrieved from the multimap based off a given month and year key. The records retrieved was then inserted into a vector. The vector was the data structure sent to the various mathematical calculation functions.

The use of a vector in this scenario had its disadvantages.

Insertions of retrieved records from the multimap into a vector was in O(n) time. So, far better data structures exist for faster insertions. Stacks, queues, binary search tree, linked list, red black tree and hash tables all have better insertion time.

The use of a vector in this scenario had its advantages.

Firstly, the mathematical calculations needed includes, calculating average, standard deviation and the sum of all the values. All of these calculations were written as separate functions taking a data structure of different numbers. For example, the total or sum of all values function takes in a data structure of numbers and adds every single number in the data structure together. In order to do this, every single number or element in the data structure had to be accessed and retrieved. This process would always be performed in O(n) time, regardless of the data structure.

Thus, even though a stack and queue allows O(1) time insertions, it is definitely not appropriate in this scenario; since every element in the stack or queue would need to be access and retrieved. Similarly, a linked list, binary search tree or red black tree would also not be the most ideal structure since the elements in those data structures are not stored in contiguous memory. This means elements could possibly be stored far away from each other in memory making retrieval of every element not ideal. On the other hand, a vector will always store its elements in contiguous memory which means every element will be located close to each other. This makes retrieval of every element easier.

So, given the vector size is known, the vector serves as the most appropriate data structure for accessing and retrieving every single element.

Role of binary search tree-

The binary search tree had some limitations. The biggest being the insertion of ordered data would result in a linked list being formed. Due to this limitation, I decided from the start that the binary search tree should store the least amount of data. In this event, a linked list would only be formed from the small amount of ordered data, thus the affect is reduced.

For my assignment, the results for the required month of a requested year were in fact ordered when inserted in the binary search tree. So unfortunately, the binary search tree in my assignment would always create a linked list. However, in my program the max number of nodes the binary search tree could have at one time is twelve, so the affect is minimal.

One benefit for using a binary search tree is that the InOrder transversal easily fulfills menu option two, three, four's requirement of outputting the results from January all the way to December. The binary search tree is also robust in that it allows the months to be outputted easily starting from December to January if required.

Another benefit, is that in the event the results aren't inserted into the data structure in order. The Bst will by default still be able to output the results in order through the InOrder transversal. So, the Bst isn't dependent on the order in which the results are inserted.

Role of KeyMap struct, KeyMultiMap class, SensorLogOutput struct and BstNode struct-

The KeyMap.h struct was used to encapsulate the key used for our map. The KeyMap.h was a struct because it didn't have any invariants. Both the data members didn't need to have protected and controlled access since they were encapsulated by their own classes.

On the other hand, KeyMutiMap.h encapsulates the key used for our multimap. This had to be its own class because the data members had invariants and needed controlled access. The month data member needed protection in that it would not accept months greater than 12 and less than 1. So, putting the month and year data member in a struct would not be suitable.

The BstNode is a struct because there are no invariant properties. Meaning neither of the three data members rely or control what can and can't the individual data members store. Using a class and having setters and getters for the node would be pointless because the setters wouldn't have any logical controls on the data members. So using a class isn't suitable.

SensorLogOutput.h was put in a struct in order to ensure the correct calculations were given to the right data members when passed to another function. Had a struct not been used, the only control, when passing the calculations to another function, would have been the order and datatype where the function was called. For example, if the function was OutputResult(int avgWindSpeed, int avgTemp) and calling function was OutputResult(avgTemp, avgWindSpeed) the function would still be called but the results output would be wrong. Using a struct controls this.

Why can solar radiation be checked in sensorData.cpp?

```
void SensorData::SetSolarRadiation(float newSolarRadiation) {

    if(newSolarRadiation >= 100) {
        m_SolarRadiation = newSolarRadiation;
    } else {
        //std::cout << "Error: Not a valid solar radiation! \n";
        m_SolarRadiation = 0;
    }


}
```

So, when the csv is read in the solar radiation is assigned and checked through the setSolarRadiation function. When an invalid solar radiation is read/set, the invalid solar radiation will default to zero. Calculating the total solar radiations (menu option 3 and 4) will not be negatively affected by this since adding zero to the total solar radiation will not change the total solar radiation.

Why can’t Windspeed or air temperature be checked in sensorData.cpp class?

```
void SensorData::SetWindSpeed(float newWindSpeed) { //SELF NOTE: Why can't we check windspeed? Because then the average windspeed would be wrong by taking a negative windspeed as 0

    //if(newWindSpeed >= 0) {
        m_WindSpeed = newWindSpeed;
    //} else {
        //std::cout << "Error: Not a valid windSpeed! ";
        //m_WindSpeed = 0;
    //}

}
```

```
void ExtractVectorOfWindSpeed(const Vector<SensorData> &SensorLogs, Vector<float> &WindSpeeds) {

    SensorData TempSensorData;
    float tempWindSpeed;

    for(unsigned i = 0; i < SensorLogs.GetSize(); i++) {
        TempSensorData = SensorLogs[i];
        tempWindSpeed = TempSensorData.GetWindSpeed();

        if(tempWindSpeed >= 0) {
            WindSpeeds.PushBack(tempWindSpeed);
        }
    }

}
```

Unlike solar radiation, both the Windspeed and air temperature are unable to be checked through their respective setters in sensorData.cpp. It needs to be checked during the calculation process or calculations will be inaccurate.

For instance-

Assume windspeed >= 100 is checked in the SetWindSpeed(…) function. All invalid windspeeds will default to a valid number 0. Thus, when we calculate the average windspeed the invalid windspeeds will influence the results which is not what we want.

By allowing the windspeed and air temperature to be checked during processing the invalid windspeeds and air temps can be excluded when calculating the averages.