CLASS DESIGN

Linked List class:

This class was used to create a linked list of nodes where each node is a time series. This class uses the following functions for various operations such as load, list, add, update, print, DELETE, biggest, deletelinkedlist.

The load function is used to populate the linked list. It takes in a parameter “country” and whatever county is passed in is what we want the linked list to me made with. In the function I loop through the lab2\_multidata csv file and for each timeseries with the passed in country name I create a new node in the Time Series class and add it to the linked list. Then the list function prints the country name that we are making the list for as well as the country code then all of the series names for each of that countries Time Series.

Add and update functions are very similar, both are passed in a series code, year, and data. They iterate through the linked list to find the a node with the matching series code. If the series code isn’t in the linked list the function returns. However, if a matching node is found I call the ADD or UPDATE function from the Time Series class, passing the year and data. Utilizing these functions from the Time Series saved by reusing the logic already implemented in the Time Series class.

Print is quite simple, it is passed a series code and prints the used the year and data for that series code. This is implemented by iterating the linked list looking for the matching series code, and if it is called I call the PRINT function from the Time Series class which then prints all of the data and corresponding year for the series code.

The delete function is used to remove a node from the linked list. It is passed a series code then I iterate the list and remove the node from the list if it is found. Deletelinkedlist is similar but instead of deleting one node it deletes the entire linked list.

Finally, the bigger function is used to find what node has the largest mean for the stored data. I iterate the list and calculate the mean for that nodes data, then I compare it with the current “biggest mean” and if the “new mean” is greater assign the biggest mean to the new mean value.

Time Series Class:

In Project 1 I implemented a class called Time\_Series. In this class I made the member functions Load, PRINT, ADD, UPDATE, mean, is\_monotonic, and best\_fit. These are the functions that are required to complete project 1. I also developed three other helper functions that were useful in some of the other functions.

The first helper function that I implemented is arr\_shift, this function takes in three parameters, int temp\_val, int year, and double data. I use this helper function in my ADD function to shift all the elements in the array over one array index to maintain chronological order of the years.

The two other functions were used to help in my LOAD, ADD, and UPDATE. The first function is called does\_arr\_need\_resized. Anytime that I am adding or removing anything from the data and year array I call this function. This function determines if I need to double or quarter the capacity of the data and year array. If resizing is needed I update the m\_arr\_size variable and call the second function arr\_resize.

In arr\_resize I first allocate two temporary dynamically allocated arrays of capacity m\_arr\_size, then I loop through my data and year array in order to populate the temporary arrays with the information stored in the data and year array’s. I then deallocated the original arrays by using delete, after calling delete I assign the data and year array to the memory address of the temporary arrays.

ALTERNATIVES AND JUSTIFICATION

For Project 2 we had to decide what data structure we wanted to use to store all of the timeseries. I chose to implement a linked list. I decided to pick this because I feel that linked lists are something that are used quite often in industry and I currently don’t have much experience with linked lists. Therefore, I thought this would be a great opportunity to get some experience with implementing a linked list.

I decided to go with a singly linked list without a tail pointer. I chose to do this over other types of linked list because I didn’t feel like having a doubly linked list was necessary. This is because I wasn’t going to be doing any bidirectional traversal and with not having much experience with linked list a felt a singly linked list would be easier to implement. The tail pointer is most useful if I am adding to the end of a the linked list often but this won’t happen in this project so I didn’t implement a tail pointer.

From project 1 we had the choice of either storing invalid data in our array’s. I decided to discard any invalid data, to me this made more sense, and I feel that this way would’ve been easier to implement rather than storing the invalid data.

In LOAD when populating the data and year array I would check if the data equals -1, if it did, I would increment the year because you need to keep track of the years for the rest of the data. Then if the data wasn’t equal to -1, I would use my private member variable m\_count to add the data and year into the arrays then increment m\_count and the year. In the add function if the user attempts to add invalid data I print failure to the console and return.

The UPDATE function is a little bit more complex since you can update a years data with invalid data. I am not storing invalid so I have to check if the “new” data that being updated is -1 and if so remove that year and data from the array. To do so I have to shift all elements back one array index then decrement m\_count since there is one less element in the array.

If I decided to store the invalid data I would’ve had to check in the PRINT function to skip any invalid data. Also in the , is\_monotonic function I would’ve had to make sure that the invalid data doesn’t affect if the data is monotonic.

RUNTIME ANALYSIS

The DELETE function would have a worst case runtime of O(N+m) assuming that with deleting a node from the linked list we also have to delete the array that is stored in the node. Assume the node we want to delete is the N node in the linked list we would iterate N times to get to that node, which has a runtime of O(N). Now assuming we also delete the array stored at that node, we would call delete[] on the array which has a runtime of O(m) since there are m items stored in the array. Since iterating through the linked list and calling delete[] on the array are not nested we would have a worst case runtime of O(N+m).  
  
The DELETE function has a worst case runtime of O(N+m), assuming that deleting a node from the linked list also requires deleting the array stored within that node.

If the node to be deleted is the N node in the linked list, we must traverse N nodes to reach it, which results in a runtime of O(N). Once we reach the node, we also need to delete its associated array. Calling delete[] on this array takes O(m) time, as there are m elements stored in it.

Since traversing the linked list and deleting the array are independent operations since they are not nested within each other, their runtimes add rather than multiply. Therefore, the worst case runtime of the DELETE function is O(N+m).

Let N be the number of nodes in the linked list, and for any given node, let there be m items of data stored in an array. Now, assume that we are trying to add data to node N for year m. The add function will have a worst case runtime of O(N+m) because we first need to traverse all N nodes in the linked list, which results in a runtime of O(N).

After iterating through the N elements of the linked list, we then call the ADD function of our Time Series class, which has a worst case runtime of O(m) when the year and data being added are at the end of the array. In this case, we assume the year is larger than all other year entries, so we reach the worst case scenario of O(m).

Since traversing the linked list and executing the ADD function in the Time Series class are independent operations and not nested within each other, the total worst-case runtime is O(N+m).