CLASS DESIGN

In project 3 I implemented two new classes, Tree, and Tree\_Node. I used the Tree class to build the tree, this class has the root of the tree and all of the functions I implemented to construct the tree. Inside the Tree\_Node class I have a few member variables that I need the trees nodes to store. Some of these are required to build the tree like a pointer to the left and right nodes. The other variables I used to store important data like the mean value range as well as the max and min index variables of the array that stores the mean values.

Then in project 5 I had to create a graph. To do this I used some of my existing project 3 functions while implementing new functions to meet project 5’s requirements.

In this project, I implemented the following functions:

1. Load Data: Loads all time series from the dataset into an array of size 512. Each index in this array stores a pointer to the head of a linked list. These linked lists are built using my linked list class from Project 2. Each list contains a country, its country code, and all time series data for that country. This is achieved by calling my linked list’s load function and storing the result at index *j* in the m\_list array.
2. Initialize Graph Nodes: Creates a disconnected graph where each node represents a different country. Each node is assigned a key corresponding to the country name, and its associated values are initialized as empty.
3. Update Graph Edges: Updates the edges in the graph based on the inputted relationship. This is done using the build tree and find functions from Project 3. A binary tree is first built for the given series code representing the relationship. Then, the find function is used to identify all countries that satisfy this relationship, storing them in a vector. Finally, the graph is updated by adding this relationship to all countries in the vector.
4. Find Adjacent Countries: Finds all countries that are adjacent to a given node in the graph. This is done by locating the specified country in the graph and iterating through all countries that share a relationship with this country.
5. Find Paths Between Nodes: Finds all possible paths from node A to node B using Depth First Search. Instead of recursion, a stack ADT is used to manage the traversal process.
6. List Relationships: Identifies and lists all relationships that two adjacent nodes share. This is done by first finding Country 1 in the graph and checking whether Country 2 exists as a value under its key. If it does, the function iterates through the relationship vector and outputs all relationships shared between these two countries.
7. Exit Program: Provides a way to exit and return from the program.

ALTERNATIVES AND JUSTIFICATION

In this project, I made several key design decisions. The first significant decision was determining how to store the time series data for all countries. The only constraint provided in the project guidelines was that the data had to be stored in an array of size 512. There were multiple ways to implement this, but I decided to create a new class called Tree.

Within this class, I defined a statically allocated array of type LinkedList\* with a fixed size of 512. This gave me an array of pointers to LinkedList objects. I chose this approach because it allowed me to easily rearrange the linked lists within the array by swapping their pointers rather than moving entire objects in memory.

If I had instead dynamically allocated the array, I would have had to use double pointers which is an approach I wanted to avoid. Another alternative was to use an array of LinkedList objects, but this would have prevented me reordering elements, because I would be moving LinkedList objects around in memory rather than moving the pointer that points to that address.

The next major design decision was how to build the tree. To accomplish this, I first created another class called Tree\_Node, which stores both the data relevant to each node and the additional information needed for constructing the tree.

Since the tree is built based on a range of mean values, I needed a way to sort the LinkedList objects in m\_list according to their mean values. To do this, I created a second array to store the mean values. I then wrote a function to populate this array by iterating through m\_list and retrieving the mean value for the specific series code being used to construct the tree.

Once the mean values were collected, I implemented a bubble sort algorithm to sort both the mean values array and the corresponding m\_list entries in ascending order. Sorting them together ensures that m\_list remains properly aligned with the mean values, which is crucial for efficient tree construction.

To actually build the tree, I wrote a recursive function along with a wrapper function to ensure the recursive function receives the correct initial parameters. The wrapper function initializes the root node and then passes the root, along with its minimum and maximum mean values, to the recursive function. The recursive function then continues building the tree by passing relevant values and populating each node’s information.

One additional design decision I made was to explicitly pass the minimum and maximum mean values at each recursive step. Since the mean values are already sorted, this approach allows me to easily determine the range of countries corresponding to each node. By identifying the indices of the min and max mean values in the sorted array, I store these positions as member variables within each node, which I use to know what countries are included in the node.

The final significant design decision I made was determining how to handle deleting countries from the tree. I chose to implement a second array of LinkedList pointers called m\_list\_del, along with an integer variable m\_items\_del, which was initialized to zero.

Whenever the user wants to delete a country from the tree, I first delete the entire tree using a post-order traversal. Then, I remove the specified country from m\_list\_del and increment m\_items\_del by one. After that, I rebuild the tree using m\_list\_del instead of m\_list. Additionally, everywhere in my code where I iterate from 0 to m\_items\_stored, I now iterate from 0 to m\_items\_stored - m\_items\_del to account for the deleted entries.

I chose this approach because removing a country directly from the tree would have been difficult. Deleting a country from every relevant node in the tree would require careful adjustments, especially when dealing with edge cases. For example, if the rightmost node contained only one country, and that country was deleted, it would require restructuring multiple nodes to ensure the next largest mean value was correctly positioned. Handling such cases dynamically would have introduced additional complexity and potential errors. By deleting the tree and rebuilding using a function that already works rather than attempting to deal with all the edge cases of restructuring the tree definitely saved me some time.

RUNTIME ANALYSIS

The runtime for my adjacent command is O(N). This is because my graph is type std::map <std::string, std::map <std::string, std::vector<Relationship>>>, where Relationship is a struct containing the series code, threshold, and relation for each tuple. In the adjacent function, I first access m\_graph[country], where country is the input parameter. Since my graph is structured as a nested map, I then iterate through the inner map to retrieve all associated country names. In the worst case scenario, there could be N countries stored in the inner map, meaning the function must iterate over N elements. As a result, the worst case runtime complexity is O(N).