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.

For this project it was required to have functions to do the following:

1. Loads all time series from the data set into an array of size 512
2. List all of the time series for any given country name as well as that country
3. Output the range (min and max mean values) for any given time series
4. Build the tree for any time series
5. Find either the mean value less than, equal to, or greater than an mean value
6. Delete a country from the tree
7. Find the countries stored in the left and right most node
8. Exit to return from the program

To complete all of these I used my code from project 2 as well as recursive functions to build and search the tree.

The load, list, and range functions were easier to implement than the rest because you don’t need to build. For load I have an array that I use to store pointers to the head of linked lists. These linked lists get build from my project 2 linked list class. Each linked list has a country and all of the data for all time series for that country. To do this I called my Linked List load function and stored the that in index j of my m\_list array. The list function I passed the country name that is wanted then loop through my m\_list array until I find the matching country name. Once the match is found I call my Linked List list function which lists all of the series codes for that country. Then for range I pass it a series code then loop through the m\_list array and check the mean value for each country that has the inputted series code. The I update the min and max means as I go through. Once I loop the entire m\_list array out the min and max means.

The rest of the required functions were more complex since all of them you need to use recursion. For build I used a recursive function to build the tree and populate all of the nodes. This function is used when the user requests to build a tree and within my delete function. In delete I recursively delete the tree then call build again to build the tree without the country we wanted to delete. For find depending on the input being less, greater, or equal I have three different recursive methods. Then for limits if the input is lowest I go to the left most node and for highest go to the right most node then output those countries.

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.

Finally, the last significant design choice I made was how I delt with deleting countries from a tree. What I ended up doing was creating a second array of LinkedList pointers called m\_list\_del as well as an integer value m\_items\_del which was initialized to zero. Then every time the user wants to delete a country from the tree I delete the entire tree using a post order traversal, then I build the remove the country from the m\_list\_del array and add one to m\_items\_del. I then proceed to rebuild the tree using the m\_list\_del array rather then m\_list. Then everywhere in my code where I loop from 0 to m\_items\_stored I actually do 0 to m\_items\_stored minus m\_items\_del.

I decided to delete countries from the tree this way because actually going through the tree and removing a country from every node seemed like a difficult task. There are also some weird edge cases that would’ve been tough to deal with. Like if the right most node had only one country and it was the country you were deleting. Then you would have to move a bunch of nodes around so that the next largest mean value would be in the correct spot.

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

To achieve a best-case runtime of O(log(NUM\_NODES)) for the LIMITS function, the tree must be balanced. This logarithmic runtime occurs because the function navigates either to the leftmost or rightmost node, depending on the operation.

If we are computing the lowest LIMIT, the function will repeatedly follow node->left to reach the smallest value. Each time we move left, we halve the number of remaining nodes in the search space.

At the root, we start with NUM\_NODES. After one left traversal, we are left with approximately NUM\_NODES / 2. After another, NUM\_NODES / 4. This process continues until we reach the leftmost node. Therefore since after each left traversal we halve the NUM\_NODES to search we get a best case runtime of O(log(NUM\_NODES)).

To achieve a worst-case runtime of O(NUM\_NODES) for the LIMITS function, the tree must be degenerate tree, so behaving like a linked list, where each node has only one child. For example, if we are computing the lowest LIMIT, and every parent node has exactly one left child, the function will have to traverse every node in the tree to reach the leftmost node.

The search starts at the root. From there, it follows node->left NUM\_NODES times since every node has only one child to its left. Because there are no shortcuts or branches to skip over sections of the tree, the total number of recursive calls is the same as the number of nodes in the tree. Therefore this would result in a worst case runtime of O(NUM\_NODES).