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 4 I had to build off of my project 3 by implementing hashing. To do this I didn’t change anything class wise, but I didn’t have to implement some functions per the project 4 requirements.

For 4 project it was required to have working project 3 functions as well as the new project 4 functions. All of these functions do the following to do the following:

1. Loads all time series from the data set into an array of size 512 (Inserting them into there index by using hashing)
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. Lookup a country code in the array and output the index and number of searches
9. Remove a data set from the array
10. Insert a dataset into the array
11. Exit to return from the program

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, country code, and all of the data for all time series for that country. To load the countries into the array I would call my helper function to find the hash value for that country code. Once I found the hash value, I would then call my load function from project 2 and store that in the array at the hash value. 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.

Functions 4-7 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.

Then functions 8-10 were all based around the hashing function. For lookup I take the inputted country code then use the hash function I implemented to look through the array and I compare the country codes. In remove I use the same logic as lookup but instead of outputting the index and number of searches it took to find that country codes location, I will delete that country from the array. Finally, for insert I again use the same hashing helper function and then insert the country into the array unless that country is already stored. If the country is already stored, I do nothing.

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 hash function used in this project is defined as array index=(H1W+iH2W) % m where H1 and H2 are computed in O(1) time, m=512, and i represents the number of collisions encountered during hashing. Initially, i=0, and it increments by 1 after each collision.

To achieve a best case lookup runtime of O(1), each element must be stored in the array without any collisions. This means that for every stored element, the initial hash value when i=0 must be unique. If this condition is met, the lookup function will always retrieve an element in best case time of O(1).

In contrast, the worst-case lookup runtime of O(N) occurs when there are N collisions. This would require every inserted element to have the same hash value when i=0, meaning that the values of H1 and H2 remain the same for all elements. As a result, the only changing factor in the hash function is i. When inserting the Nth element, the function will increment i up to N times before finding an empty spot, leading to an insertion time of O(N). Similarly, when looking up the last inserted element, the hash function must be evaluated N times, resulting in a worst case lookup time of O(N).