**ASSIGNMENT 4 REPORT**

By: Khadija Swailem 202301222

***Assumptions:***

Assumed that tree is of infinite size and assumed we have infinite memory to occupy the large size of the data set provided to test our code with .

***Design approach:***

In this assignment I had many ideas and the I documented the changes in a github link at the end of this document as well as the cpp files of the attempts.

My general thought process was to create a hashtable with prime size to minimize collisions and allow even distribution across the table to prevent the table being full in one part over the other. I chose the size 31.I also chose the hash function that uses horners law to allow even distribution, I will discuss my choice of hash function in more depth in ***Hash Function choice***. I wanted to make each index in the hashtable have a pointer to the AVL tree where we will insert the weblogs.Then I wanted each line being read from the file to be created as an object and then inserted into the tree in specific index based on the hash value of hashing that weblog IP.If the index the hash function returned (hashvalue) was empty(no tree so pointing to null) I will create a tree and insert in it the weblog according to date and time and hence I used the class myData that I had previously created that overloads the operators <,> to be used in inserting according to the date and time of the existing weblog in the tree and the weblog we want to insert.I also wanted to read the IP I wanted to calculate a hashvalue for directly to be able to know which index will I insert/create a tree in.

***Hash Function choice:***

I chose the hash function provided in the slides that adopts horners method because first of all it hashes all the characters in the IP and takes into consideration the position of each character by using horners law and calculating a polynomial function.The constant 37 is multipilied by each character in the IP so each character will now be considered a constant of the polynomial equation. Due to the presence of a weight (37) which is also prime this function is expected to distribute evenly across the hashtable and avoid patterns that could lead to clustering which we don’t want in the hashtable. Although one draw back is that it is slow with long keys but since the complexity of it is O(n) where n is the number of characters in the key,but since the ip addresses we are hashing are only between 10-12 characters long, then n’s value is between 10-12, which is relatively small so it won’t be as slow.

**Approach 1:**

In **inefficient attempt.cpp** I was using a primary and kind of inefficient approach which is:

1.I wanted to make a function that reads each IP address being inserted and returns it so I can calculate its hashvalue using the hash function I chose and then know what index I can insert/create new tree and insert.So I created a function called **readIP** that opens the file reads only the IP and then inserts it into an array called IP which then is looped by and external index so that each time function is called the following IP in the array will be returned starting at zero, and since I increment the index before the return statement of the array content so that its executed I initialized the external index variable to -1 so that the first time its at index 0(first element in the array).

2.I followed the same approach in the **fileopener** function which does the same as **readIP** but reads all the file contents and creates objects from them and stores them in an array then also returns the array content with the same method as **readIP** using an external index.

3.In the main,first, I tested the array to actually check that it will work. Then I created a hashtable of type AVLTree with the template of class myData which I the class with the overloaded operators for date and time for the file objects,its size is 31 to be prime to allow maximum even distribution and to minimize collisions.I initialized its indexes with NULL.I then calculate the hash value by calling **hash1** and passing its parameters as readIP("log20170630.csv") to return the ip at index were at every time its called depending on the call number (e.g first time index =0 so element at 0) and also the size of the hashtable 31.Then I check if the hashtable of hashvalue calculated is NULL (meaning still empty so ==NULL) the I create a tree and call **insert** with parameter fileopener("log20170630.csv") which will also return the object at the index were at upon call number.Else if index wasn’t equal to NULL then a tree already exists then will call **insert** with the tree of that hashvalue index and also pass to it fileopener("log20170630.csv") to be inserted.Finally I attempted to display the tree by calling **traverse** for each index of the 31 indexes of the hashtable.

However, although I had the correct thinking the way I implemented it was inefficient because calling the function each time to just get one value would mean that, for both, **readIP** and **fileopener**, they would iterate over the whole file read it and restore it in the array then get the ip/weblog at the index were currently at.With my test case of 1000 there wasn’t an issue but with the enormous size of the data set it would take ages to run and its now apparent that this is not the most efficient approach so I used another approach.

**Approach 2:**

In **assignment 4.cpp** I used a more optimized approach.In the second approach I used the same base as approach 1 but I changed the functions **readIP** and **fileopener** ‘s return type to void and declared two arrays in the main, IP of type string which will store the Ips being read and arr of type myData which will store the weblogs themselves. I then passed these arrays to the **readIP** and **fileopener** so that the ip addresses and the weblog objects can be stored in them and since arrays use pointers they’re automatically passed by reference to the function and when we edit in it inside the function its edited everywhere including in the main scope so in the main I could directly call the **readIP** and **fileopener** once with both the arrays which will store the values and then I could loop over the arrays and extract the weblogs to be inserted as well as the ip address were trying to hash.This saves time instead of calling the **readIP** and **fileopener** each time and reading the whole file every time.Then I thought of putting the array that stores the ip addresses in the same loop as the objects and discarded **readIP** completely.I tested the code still on the 1000 values test case and I traversed the tree by calling the AVL **traverse** function for each hashtable index and passing the root of that index to the **traverse** so that its not automatically default set by NULL.I also made a check to make sure that there is a tree at the hashtable index to prevent returning NULL and I display an error message then continue so that the for loop can move on to the next iteration (next index).

***Method for searching a record with specific IP within specific date and time range:***

For the IP search function within a specific range I added some changes to the **rangesearch** function that I had already implemented in the past assignment in the BinarySearchTree class.I added more parameters to the function so now it takes the lower and upper bounds of the date and time range its supposed to search within along with the key (IP address) it should search for and along with the root of the tree at the specific hash index where its supposed to be located.I faced an error that was an issue with the operator = = because the **rangesearch** function couldn’t compare the key so to overcome this issue I overloaded the = = operator in the myData class as I did before with the > and < operators last assignment according to date and time.The new **rangesearch** function of the BST class compares the item’s ip to the key that’s given and also compares the item’s date and time to the given bounds of date and time and recursively calls itself with the right/left subtree according to the result of the if statement(it follows the same logic as the original **rangesearch** function).Theres also an additional case that if the date and time of the node selected are within range but the ip doesn’t match the ip we are searching for then it displays an error message and returns NULL. Inside the **IPsearch** function I calculated the hashvalue of the key were looking for to know which index tree will we look at and then search inside it.I also assigned the root of that tree to a node n so I can pass that root to the **rangesearch** function as the root of the tree at that index.Then I call the **rangesearch** with the tree at the index (hashval) and pass the parameters to it.The **IPsearch** also handles the case that the tree at hashvalue index doesn’t contain the ip address were looking for it should return NULL. I tested the function in the main for case of being found and not being found, the **rangesearch** function returns the address or NULL if it wasn’t found.

***Big-O Analysis:***

**Hash table Complexity**: The complexity of the hash function used to hash the ip addresses into the hash table is O(n) where n is the length of the key but since we know that the ips length is assumingly small and constant we can consider it as O(constant).Insertion has complexity of O(log n) because AVL trees self balance, but since we loop the number of times as the number of rows in the file then its O(n).Then overall the complexity of the hash table is O(1)+O(log n)+O(n) if we discard the time complexity of calculating the hash value and time complexity of inserting then time complexity is O(n).

**IPsearch Complexity:** Since this function calls the **rangesearch** we have to put its complexity into consideration.The complexity of the **rangesearch** is O(log n) because the recursion cuts half of the tree and searches it each time and since the tree is guaranteed to be balanced because its an AVL tree then the worst case of having a linked list tree won’t happen.Since the rest of the operations are simple they have a complexity of O(1) which is ignored so the overall complexity of this function is O(log n).

**Why is this hash function provides the best running time:**

It’s the best running time for this application because I used benchmarking to measure the time taken for this program to run on a sample size 1000 using different constants other than the 37 for the hash function, which on average had running time 3.04. Constant 41 had average running time 3.31 and constant 33 had average running time 3.24.Hence, 37 is the best value for the weighting in the hash function for this application. These values aren’t 100% accurate though due to reaction time on my end to start and stop the stopwatch as well as my laptops overall performance that’s why I ran several times and took the average.

NOTE: I USED A BREAK IN THE WHILE LOOP TO BREAK AFTER A CERTAIN NUMBER OF ITERATIONS AND USED A SAMPLE SIZE 1000 FOR THE EASE AND EFFICIENCY OF TESTING.

***Extra:***

I pushed my code to github under repo name hashtable and function for ip where I commit the changes as I go and heres the link: <https://github.com/khadijaswailem/hashtable-and-function-for-ip>