To get the number of operations for the addition of a new node you would have to add in the new data as a leaf node, compute the hash of the leaf nodes to be the parent nodes it would take the number of leaves you are inputting to insert new data into my nodes.

To compute the hashes for everything upon insertion of a new node we would have to then add the new nodes in as leaf nodes under the parent nodes and recomputed the hashes of the parents based on the new information in the leaves we added. So for every new leaf we would have to do computations that are the size of the tree + number of leaves submitted. However you could make it go faster and just insert the data without hashing the new nodes and then after they are inserted we could hash everything and that way we aren’t wasting operations hashing.

To compare if 2 trees are identical we would only need to do computations equal to the total size of the tree because we would be able to just test the hashes (assuming there’s no collisions) and see if the hashes come out the same then the trees would be equal. We cant just test the root because the hash of roots kid nodes could be the same but the kids of those kid nodes could be different so we would have to check the entire tree.

Determining where the two trees differ is also going to be the size of the tree in operations as a worst case scenario. After running the test to see if the trees are equal we would have to then run a variation of the same test to see where they differ, checking the hash results we would be able to tell that at a certain node the hash returned is different from the other trees returned hash at the same node. This is why it might be the size of the tree in operations and it also might be a lot less. You don’t know where the difference is to say when the number of operations ends.

For my first hash function (hash\_1) is one of the simpler hash functions as we are simply taking an integer value and adding it to the number we wish to hash in the place of the array increasing up until the size of the number that we are adding to the tree.

For the second hash function (hash\_2) its also kind of simple but a little more complex than the first one. We are taking a number (sum) and adding it to the size of itself + the size of the array at the index that is being incremented until the length of the number that we are inserting has been reached. And then using the modulo operator and dividing by 307 (area code). We could divide by the size of the vote but the votes are all the same size so it would work out to being a constant number anyways. Or set 307 as a global size variable but just using modulo on 307 was easier.

For the third and final hash function and the most complex of the three I am setting a variable to be 131. I then take a variable initialized to be 0 and am adding it to itself while also making itself equal to the number at the vote we want to inserts data at array index I. Going up until the array index has reached the size of the vote that we are submitting. I am then multiplying this number by the 131 we created earlier and lastly doing the modulo operator again with 307 to return the hash. This one works well in my opinion because the numbers at the specified arrays will always be different so the amount of collisions we will have should be minimal unless the numbers somehow align.

There is some weird error in my compiling that is saying that the functions you provided with the operator overloading of the ostream must take exactly one argument but they are designed for 2. I tried to google it and it said that the reason was that it was actually trying to take 3 arguments which was one too many from the two and that you needed to create an output helper function and do all that. I did all that and it worked a little for the bTREE but not at all for the pMT so I went to a programmers server on Discord and the consensus was to declare the function outside the class in bTREE and pMT and then in the cpp files make them friend functions but that too did not work. For now they are commented out so the program can run effectively. The commented functions include both operator<< functions in bTREE and pMT and the outHelp functions in both bTREE and pMT.