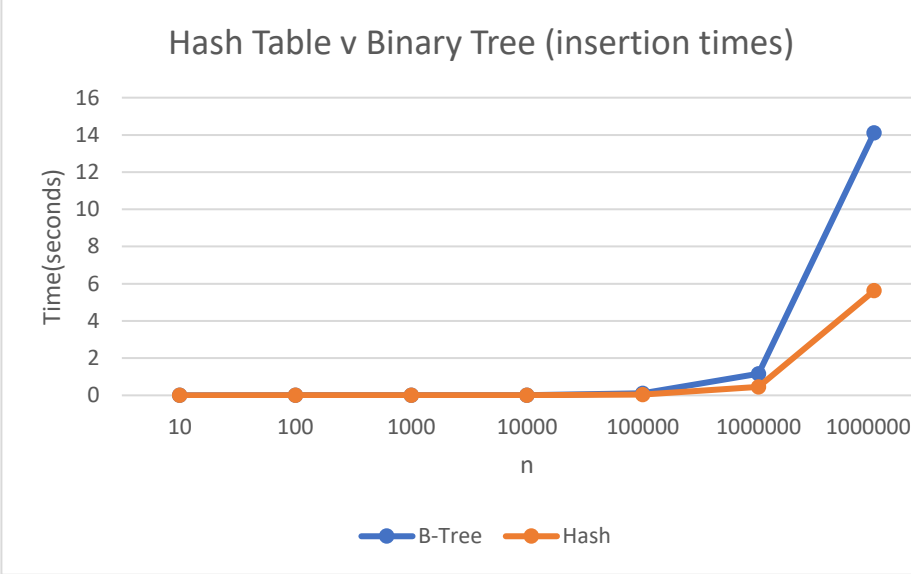
Hypothesis: Inserting into a binary tree will always take longer than inserting into a hash table. As long as the hash function is good and there are minimal collisions.

Methods: : I am using the NetBeans software version 8.2, I do not know what flags they use at compile time, I only cleaned, built, and ran the project. I had a loop for inserting into a multiset and another for inserting into a unordered multiset. For each loop, I iterated from 0-n and had the inserted the index of the loop into the multiset. I used n's: 10, 100, 1000, 10000, 100000, 1000000, and 10000000 and for each n, I compiled once and ran 10 times and averaged out the times. For the smaller n's, to get a non-zero time, I put the insertion loop into a loop. For n=10, I looped 10000 times. N=100, I looped 1000 times. N=1000, I looped 100 times. N=10000, I looped 10 times. And the rest I "looped" once.

Results:

n	B-Tree	Hash
10	0.00000411	0.00000591
100	0.0000593	0.0000643
1000	0.000725	0.000566
10000	0.00774	0.00605
100000	0.1118	0.0503
1000000	1.1623	0.4542
10000000	14.116	5.62421



Discussion: The graph is not very helpful to interpret because most of it is not visible, but the data table is very interesting. First off the reason the time for each increases by a factor of 10 for each n is because I looped the insertions. So for n=100, I inserted 1, then 2, then 3, and so on until 100. This loop is linear. It is known that insertions for balanced binary trees is big theta(log(n)) and the data reflects that, but insertions into hash tables is known to be big theta(1) or constant. This would imply that inserting into a hash is always going to be faster than inserting into a binary tree. However, for n <1000, inserting into a binary tree was faster. This is also counterintuitive because inserting into a hash isn't always constant because of collisions, and the larger the n the higher chances of collisions, which would make you expect higher times for larger n's, but time stayed fairly constant for hashes.

Conclusion: Inserting into a binary tree takes longer than inserting into a hash table, but for n>1000. For n<1000, it is faster.

```cpp
#include <time.h>
#include <ctime>
#include <cstdlib>
#include<iostream>
#include<string>
#include<vector>
#include<set>
#include<unordered_set>

int main(int argc, char** argv) {

    srand(time(NULL));


    //multiset<int> b_tree;
    vector<int> vec;
    vector<int> random_numbers;

    constexpr int SAMPLE_SIZE = 1000000;
    constexpr int NUM_LOOPS = 1;
    constexpr int RANGE = SAMPLE_SIZE*10;

    //Random numbers to insert into vector and multiset
    for (int i=0;i<SAMPLE_SIZE;i++){
        int random_number = rand() % RANGE;
        random_numbers.push_back(random_number);
    }

    //Binary tree insert
    clock_t s_time = clock();
    for(int i = 0; i < NUM_LOOPS; i++){
        multiset<int> b_tree;
        for(int j = 0; j < SAMPLE_SIZE; j++){
            b_tree.insert(j);
        }
    }
    clock_t f_time = clock() - s_time;
    cout << "Multiset Insert Time: "
         << ((double) f_time) / (double) CLOCKS_PER_SEC
         << " seconds" << endl;

    //Hash insertion
    s_time = clock();
    for(int i = 0; i < NUM_LOOPS; i++){
        unordered_multiset<int> h_tree;
        for(int j = 0; j < SAMPLE_SIZE; j++){
            h_tree.insert(j);
        }
    }
    f_time = clock() - s_time;
    cout << "Unordered Multiset Insert Time: "
         << ((double) f_time) / (double) CLOCKS_PER_SEC
         << " seconds" << endl;
```