Benjamin Miller

Binary Tree vs. Hash Table

Hypothesis:

I think that I will find that the hash table is much more efficient. I believe that I will find this because you do not need to sort when inserting new numbers. I think that between 1,000,000 and 10,000,000 insertions the binary tree will cross the 3 second mark, and the hash table will cross somewhere over 10,000,000.

Methods:

For this experiment I used CLion by Jetbrains to run the code. I created a new project in c++ 14 and put my code into the file that ended with ".cpp". I did not do anything special for the compiler or any optimizations. Once the code was added, I simply hit the run button. For the code in the main function I did these steps:

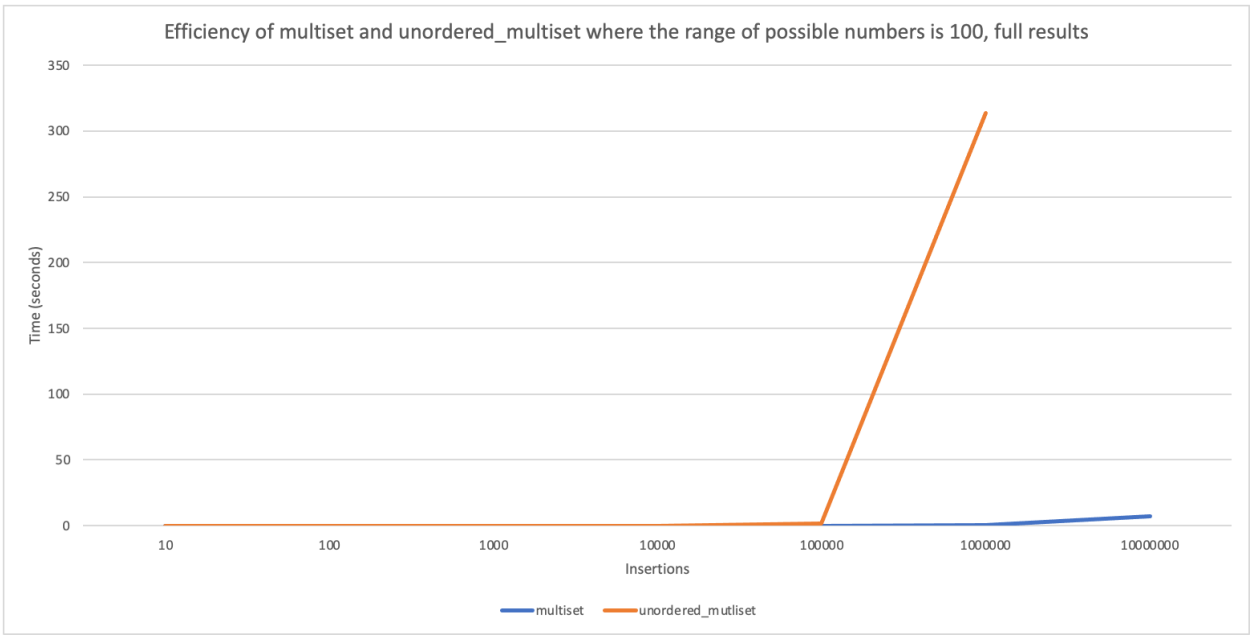
1. Define a variable for n number of insertions, a multiset, and an unordered multiset
2. Start the timer and make a loop for the total number of n insertions
3. At the start of each loop, get a random number 1-100
4. Insert the random number into the multiset
5. Record the time to complete all insertions, and increase the number of insertions until the time is over 3 seconds
6. Repeat steps 4 and 5 with the unordered multiset
7. Bonus! Repeat steps 3 through 6, but with random numbers between 1-1,000,000

The auto generated makefile:

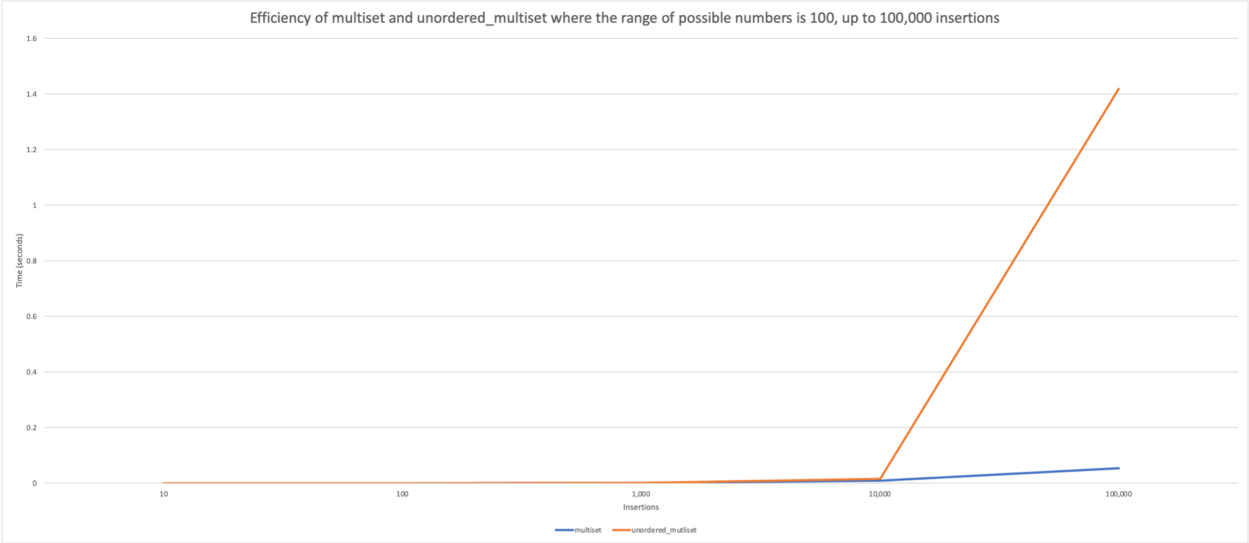
```
cmake_minimum_required(VERSION 3.17)
project(431question3)

set(CMAKE_CXX_STANDARD 14)

add_executable(431question3 main.cpp)
```

```cpp
#include <iostream>
#include <set>
#include <unordered_set>
#include <iterator>

using namespace std;

int main() {
    int insertions = 1000000;
    multiset<int> multisetTest;
    //unordered_multiset<int> unorderedMultisetTest;

    std::clock_t start_time = std::clock();

    srand(time(0));
    for(int i = 0; i < insertions; i++) {
        int randomNum = rand() % 100;
        //cout << randomNum << endl;
        multisetTest.insert(randomNum);
        //unorderedMultisetTest.insert(randomNum);

    }
    std::clock_t tot_time = std::clock() - start_time;
    std::cout << "Time: "
              << ((double) tot_time) / (double) CLOCKS_PER_SEC
              << " seconds" << std::endl;


    return 0;
}
```

Results:

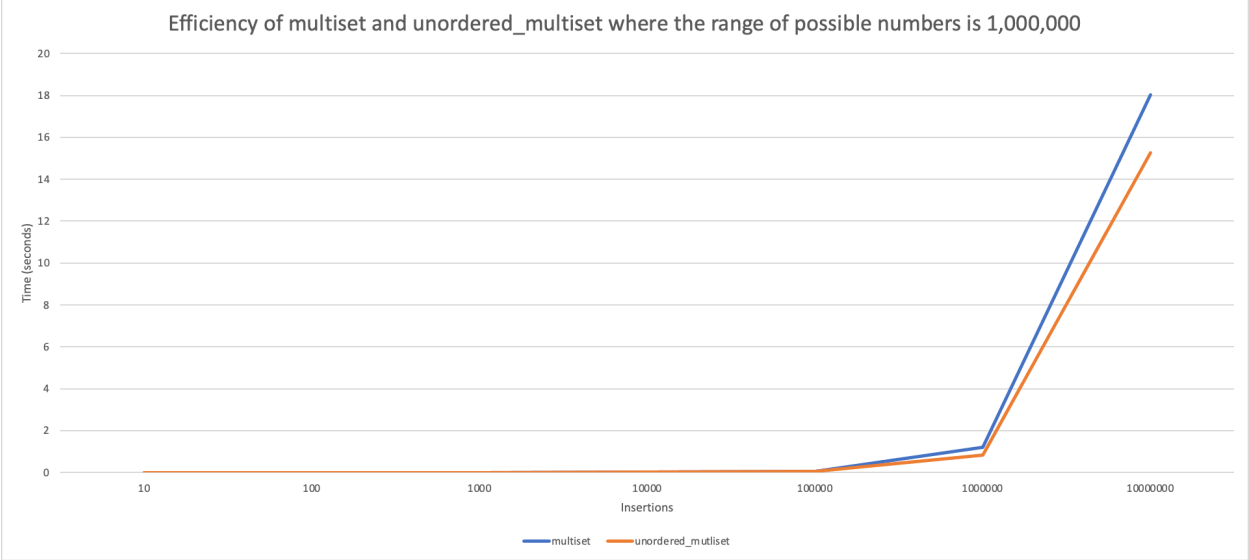


The results of the experiment for when the possible numbers to be inserted range from 1-100. The graph shows the time it takes for a multiset (blue) and unordered multiset (orange) to perform n insertions.



The results of the above graph up to 100,000 insertions. This graph shows the data before any structure is cut off for running too long.

Bonus Results:



The results of the experiment for when the possible numbers to be inserted range from 1-1,000,000. The graph shows the time it takes for a multiset and unordered multiset to perform n insertions

Discussion:

I was very surprised how badly the unordered multiset originally performed. For 1,000,000 insertions it took over 5 minutes compared to the multiset which took seconds. I looked more into why this might be happening and I realized that hash tables perform very badly when they have to insert a value already inserted. I decided to increase the range of random numbers from 1-100 to 1-1,000,000. This then gave me

results that I had originally expected. A challenge I had was the whole process of figuring out why the hash table did not perform as expected. I decided that the bonus data provided a better answer to the question of how much faster a hash table is compared to a binary tree. However, I think it is important to realize that hash tables are not faster in all situations. Overall, I was right about my prediction about the binary tree, but only partially right for the hash table.

Conclusions:

Under the conditions tested, when the range of numbers is 1-100, the multiset or binary tree is much faster than the unordered multiset or hash table at all numbers of insertions. However, for a larger range of possible values, such as 1-1,000,000, the unordered multiset is more efficient, about 20% faster, for all numbers of insertions.