Question 1:

Binary Search Trees vs. Hash Tables:

In the C++ standard library, std::multiset can be used to easily store values in a balanced binary tree. Likewise std::unordered\_multiset can be used to store values in a hash table. In theory, a hash table should be faster for insertions and deletions, but how much faster? Compare the two for insertion of n=10, n=100, n=1000, etc, -- as high as you need to go for at least one of the data structures to take longer than the other to run. Draw a graph comparing the performance. Given these results, why would you ever want to use a balanced binary tree? This problem is more easily investigated in C++; however, if you'd like to use another language, then make sure to find an implementation of a balanced binary tree.

----------------------------------------------------------------------------------------------------------------------------

**Hypothesis:**

Since in theory a hash table would have O(1) time complexity for insertion and deletion assuming the hash function is very good, we would guess that the insertion and deletion time for the hash table would not change despite increase to n. On the other hand, a balanced binary tree requires O(log n) for lookup, and O(log n) for rebalancing the tree. This is true for both insertion and deletion of a balanced binary tree. Therefore, the execution time of the insertion and deletion on the balanced binary tree would increase by 2 \* log n.

If we consider the time of insertion and deletion of the hash table as the constant factor C, then the difference in time complexity would be

T(1)  - T(n) = C  - 2 \* log n

Where T(1) = the execution time of insertion/deletion on the hash table

And T(n) = the execution time of the insertion / deletion on the balanced binary tree

**Methods:**

Since time complexity is based on n which is the size of the data structure we must test each problem on varying sizes of the data structure. For values we use integers chosen randomly using c++ rand() seeded using the c++ srand() function, and the range of values [0,100). In order to remove the initialization time of the entire program we measure only the time of execution of the insert / delete function with the varying data structures. Sizes for the data structure are

N = {10^1,10^2,10^3,10^4,10^5}

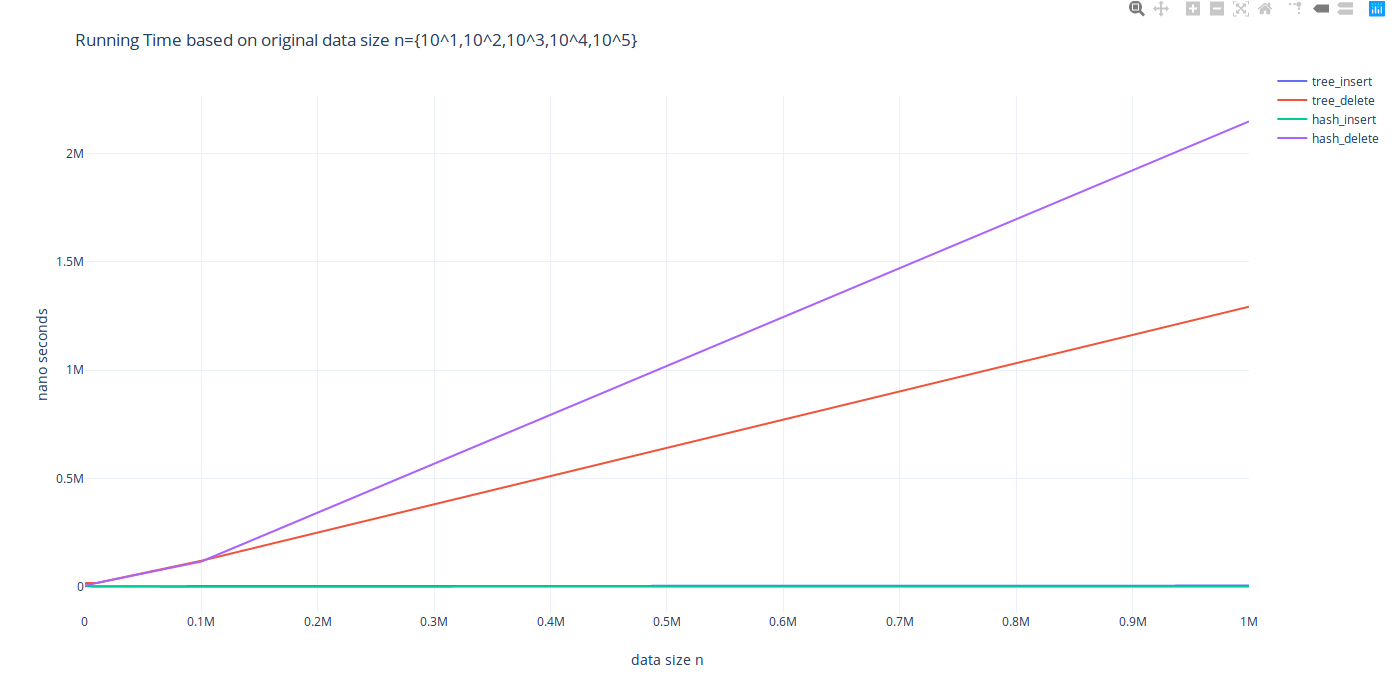
In order to test insertion we run the insertion function on a pre-initialized data using the binary tree structure of size n using the method above. We use the same technique for the hash table. We then repeat the same steps for both except using the deletion function instead. Averages of 5 iterations on

All code was created in c++ and was compiled using the statement:

g++ -std=c++11 q1.cpp -o q1.out

**Results**:

Unfortunately times on the insert on the binary tree and the hash table got stuck together. Therefore, I include the raw data points which construct the graph below, with a key on how to read it.



B\_ins = Insertion on Balanced Binary Tree

B\_del = Deletion on Balanced Binary Tree

H\_ins = Insertion on Hash table

H\_del = Deletion on Hash Table

N = Elements in the data structure before operation starts

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | B\_ins | B\_del | H\_ins | H\_del |
| 10 | 1990 | 10444 | 9137 | 1729 |
| 100 | 3071 | 7983 | 3003 | 1952 |
| 1000 | 3894 | 17449 | 3125 | 5703 |
| 10000 | 2045 | 16986 | 871 | 17482 |
| 100000 | 3184 | 120093 | 596 | 116528 |
| 1000000 | 5499 | 1.293e06 | 1074 | 2.14882e06 |

Insert on Balanced Binary Tree: constant

Insert on Hash Table: constant

Delete on Balanced Binary Tree: scales with n

Delete on Hash table: scales with n

**Discussion**:

Insertion on the balanced binary tree and insertion on the hash table both indicate constant time operation.

Deletion on the balanced binary tree and the hash table both indicate a running time with linear scale to the input.

 If we look at the c plus plus reference it states that erasing an item within the multiset (balanced binary tree) is logarithmic in container size + linear in the number of elements removed if we were removing an item by value, and amortized constant if we were removing an item by position. Since we only remove a single item ever in our test, the resulting time complexity should be log n. Insertion on the multiset seems constant from the graph, but upon closer inspection of the values we can notice that it is logarithmic in scale to the input, as it increases by about 1000 ns every time we increase the input size by ten times.

Deletion on the hash table was linear to the input size. This is completely different from our original hypothesis which considered this to be constant time. From the c++ references it states that the average case for the deletion operation on the hash table is constant whereas the worst case is n. Oneway we might achieve slower than constant speed using the hash table is if we are rehashing the table for some reason. The c++ references suggest that deletion on the hash table may cause this, but does not indicate the exact reason for so. On the other hand, insertion onto the hash table was as expected constant despite change in n.

Differences in the running time between insertion on the data structures is log n to constant time. Differences to the running time between deletion on the data structures indicates that the hash table is faster during earlier sizes of n and becomes slower in larger sizes of n.

**Conclusion**:

Unexpectedly, measurements on the deletion on the hash table seems to indicate that the deletion operation is not constant time, whereas the insertion is. The balanced binary tree performed as expected, showing logarithmic time for insertion, and linear time for deletion. Further investigation is likely required to find the extent of the why deletion on the hash table causes such poor running times. My current guess is due to the rehashing implementation and how the system determines it necessary.

<CODE BASE>

|  |
| --- |
| **#include <iostream> #include <ctime> #include <set> #include <unordered\_set> #include <stdlib.h> #include <chrono>    std::multiset<int> make\_set(int n){  std::multiset <int> S;  for(int i = 0; i < n; i++){     S.insert(rand() % 100);  }       return S; }  std::unordered\_multiset<int> make\_unorderedset(int n){       std::unordered\_multiset <int> S;  for(int i =0; i < n; i++){     S.insert(rand() % 100);  }       return S; }    int main() {  //create our data structures  srand (time(NULL));  int N;  std::cin >> N;  std::multiset<int> S = make\_set(N);  std::multiset<int> S2 = S;  std::unordered\_multiset<int> S3 = make\_unorderedset(N);  std::unordered\_multiset<int> S4 = S3;  double ns;  int x = rand() % 100;       //count the execution time for multiset insert  auto begin = std::chrono::high\_resolution\_clock::now();  S.insert(x);  auto end = std::chrono::high\_resolution\_clock::now();  ns = std::chrono::duration\_cast<std::chrono::nanoseconds>(end-begin).count();  std::cout << "bintree\_ins: " << N << " " << ns << std::endl;       //count the execution time for multiset delete  begin = std::chrono::high\_resolution\_clock::now();  S2.erase(x);  end = std::chrono::high\_resolution\_clock::now();  ns = std::chrono::duration\_cast<std::chrono::nanoseconds>(end-begin).count();  std::cout << "bintree\_del: " << N << " " << ns << std::endl;       //count the execution time for hashtable insert  begin = std::chrono::high\_resolution\_clock::now();  S3.insert(x);  end = std::chrono::high\_resolution\_clock::now();  ns = std::chrono::duration\_cast<std::chrono::nanoseconds>(end-begin).count();  std::cout << "hashtab\_ins: " << N << " " << ns << std::endl;      //count the execution time for hashtable delete  begin = std::chrono::high\_resolution\_clock::now();  S4.erase(x);  end = std::chrono::high\_resolution\_clock::now();  ns = std::chrono::duration\_cast<std::chrono::nanoseconds>(end-begin).count();  std::cout << "hashtab\_del: " << N << " " << ns << std::endl;       //for(int i = 0; i < N; i++){  //      //}       return 0; }** |

**References**

<http://www.cplusplus.com/reference/set/multiset/erase/>

For the first version (erase(position)), amortized constant.

For the second version (erase(val)), logarithmic in container [size](http://www.cplusplus.com/multiset::size), plus linear in the number of elements removed.

For the last version (erase(first,last)), linear in the distance between *first* and *last*.

<http://www.cplusplus.com/reference/unordered_set/unordered_multiset/insert/>

Single element insertions:

Average case: constant.

Worst case: linear in container size.

Multiple elements insertion:

Average case: linear in the number of elements inserted.

Worst case: N\*(size+1): number of elements inserted times the container size plus one.

May trigger a [rehash](http://www.cplusplus.com/unordered_multiset::rehash) (not included).

<http://www.cplusplus.com/reference/unordered_set/unordered_multiset/erase/>

For the first version (erase(position)), amortized constant.

For the second version (erase(val)), logarithmic in container [size](http://www.cplusplus.com/multiset::size), plus linear in the number of elements removed.

For the last version (erase(first,last)), linear in the distance between *first* and *last*.

<http://www.cplusplus.com/reference/set/multiset/insert/>

For the first version ( insert(x) ), logarithmic.

For the second version ( insert(position,x) ), logarithmic in general, but amortized constant if *x* is inserted right after the element pointed by *position*.

For the third version ( insert (first,last) ), Nlog(size+N) in general (where N is the distance between *first* and *last*, and size the [size](http://www.cplusplus.com/multiset::size) of the container before the insertion), but linear if the elements between *first* and *last* are already sorted according to the same ordering criterion used by the container.