Introduction

My design is to model a loot table that contains item numbers to identify the item and the quantity that will be dropped, using a HashTable<string, unsigned int> model. Hash tables have good lookup capabilities and when you are killing monsters or finishing a mission having the ability to quickly decide what items need to be dropped is important.

Design Philosophy

The main priority when it comes to a loot table is speed. A search function on a hash table is quick and essentially O(1), in most scenarios. This system would be used in the game so the client of this multiset would be the game system itself and the developers, and the users would be the players. Because of this, readability is less important as the user will not really need to know the full extent of what is going on in the background, just the client, and at most they will only need the ability to see what can be dropped and the chances of the drop. This system also allows the extension of the table as need be. The table insertion can be a little inefficient, but at the same time this will most likely be done in an update and not done live so the player would not experience this issue at all. The process would be simple, a creature dies or a mission completes, for this example we can say it simulates a random dice roll. This will cause some numbers to be rolled more than others. Then using that random number the program can hast the table to find the item and quantity stored there. This can be used for quest completions as the variability of the drops is usually a lot lower. For smaller monsters that will have more variance in the items dropped a single random number can be generated and hashed into the monsters table to find the item. If the variance of smaller monsters is required a random number generator can be used to generate the number of items the monster will drop in a range after the item has been located in the hash table. Because of the speed that a hash table can search for the item that needs to be dropped, I feel it is the best fit for the basis of the system's multiset.

Core Operations

The core functions of the hash table are fairly simple. Aside from the creation of the hash table the search function is a plainly obvious one as the multisets function is to do a lookup. It simply uses a random number to hash through the table to find the desired item drop. This is almost the entire reason for using a hash table as the search function will likely function with a time complexity of O(1). This isn't always the case though as there will be edge cases that change the time complexity though. The first case is if the hash function takes us throughout the entire table causing a worst case scenario of O(n). Loot tables aren't massive in size so this isn't a massive issue and will more often than not cause a complexity of O(1).

There is also an add and a remove function. These have similar time complexities to the search function being around O(1). The add operation adds a loot drop to the table and the remove function does the opposite, removing an item from the table. Both functions also have similar edge cases as the search function, remove could have to hash through the entire table to find the key that it is looking for or not find it at all. The add function would also have the possibility to iterate through the entire table looking for an open bucket. The hast table would facilitate this to not happen because of how hash tables want to maintain their weight. These functions although would not take time complexity into consideration like search would because these would be done in an update and thus the user would not have seen it.

The add function's time complexity does have another edge case though that causes another shift in time complexity and that is the rebalancing of the hash table. This causes a massive change in the time complexity of the add function. This is because the rebalancing function has a time complexity of O(n + 1) because it requires to create a whole new hash table and then fill it with the previous hash table's contents. Luckily though, this function has no edge cases as it only doubles the size of the hash table and there will never be another possibility. This is the biggest constraint that a hash table has when it comes to the core operations though. Because a hash table has a set size it can't grow and shrink as needed but loot tables don't grow or shrink in real time, thus once again this becomes something changed in an update and not seen by the user.

Because of the rebalancing function there is another function that is required. The weight function "weighs" the table to determine how likely an insertion of an item would cause a collision. This cuts down on the time required for adding so you are not going to search all 8 buckets in a table, only about half would actually be searched because it would have been rebalanced by then. The weight function just checks the number of filled buckets and compares them to the total bucket count thus the time complexity is O(1). This operation is required for hash tables because it allows the table to grow and not get too clogged with data signalling the rebalance operation to fire. This is also another operation that the user will never see happen so the time complexity can be a little more neglected.

Set Operations

A functional set-like operation would be an item lookup that would be a unique_Drop(string) function. This function would take in some kind of data, for example a location,
and return a list of loot drops that are unique to that location. This would be a more complex operation as it would compare all items with all items in all locations. This would lead to a time complexity of O(n^3). This is less than ideal, but this could also be saved by doing it when the game launches or in an update that would save it in a table. The table could then be viewed by the user when they wanted to and this function would have a time complexity of O(n) as it would only have to display the table. This wouldn't have to manipulate any data though just iterate through the desired location and then at each bucket check the other locations for a bucket with the same item id. If none are found, add the item to the unique table.

```
vector<string, string> uniqueTable;
```

```
for(all items in location){
    currentItem = location[x]
    unique = true
    for(all locations){

        for(all items in nextLocation){
            testItem = nextLocation[z]

            if(currentItem == testItem){
            unique = false
            }
        }
    }
    if(unique){
        add currentItem and location to unique table
    }
    else{
        do nothing
    }

}

return uniqueTable
```

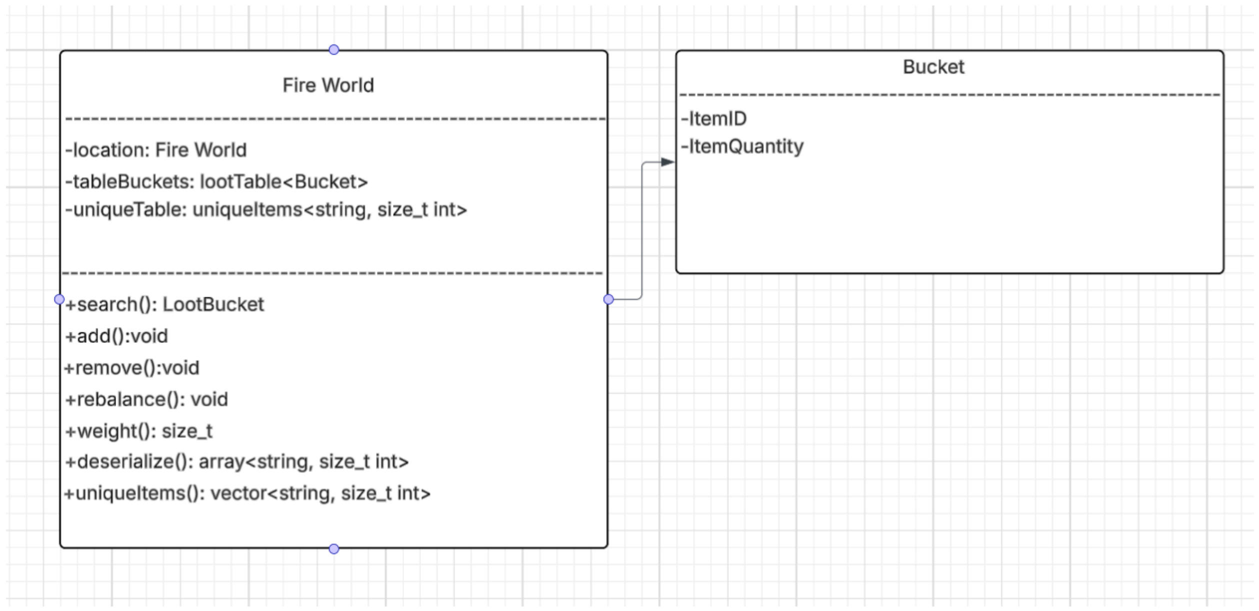
The pseudocode above gives a small example of how this code would look iterating through all items in all locations and doing a simple check in each to determine if the item is unique. If another Item is not found then the item is added to the table and once all locations have been visited the table is returned. This would take a lot of time though as stated previously, but this can be mediated with it being done as the game launches or in an update and then the table is stored in a file to be accessed easily later.

Extension Feature

A possible extension to the multiset would be the ability to deserialize() the tables. The ability for the user to load the loot table of a location would allow them to have more information available to them. In games where loot tables are the deciding factor, it's much nicer for the data to be readily available without the use of outside sources. People with two monitors will normally have a games wiki up on another screen so that they can access information easier and this would mitigate the need for such tools. This wouldn't require that much of a difference in the code. The program would have a method called by the user to request the locations loot table. The method would deserialize the data into a readable format for the user and display the table.

This can be done by creating a new array and returning that, which has the items that can be dropped from the location. The only new data that would need to be created is the array, and the new methods would simply be the call method and deserialize().

UML Diagram


Fire World
-location: Fire World
-tableBuckets: lootTable<Bucket>
-uniqueTable: uniqueItems<string, size_t int>
+search(): LootBucket
+add():void
+remove():void
+rebalance(): void
+weight(): size_t
+deserialize(): array<string, size_t int>
+uniqueItems(): vector<string, size_t int>
Bucket
-ItemID
-ItemQuantity

Trade-off Analysis

The clear alternative to the hash table would have been the AVL tree, but there were some reasons that I felt the hash table was a better fit. The hash table is a little faster on average for a search method in the scale that a game would use it for. Most loot tables are small and don't have hundreds of items in them. The AVL Tree has a worst case scenario of O(log n) which is much better than O(n) but for the scale that I am using the hash table is faster simply on average. The avl tree would also be quicker to update as well which is worth mentioning. Adding or removing items would also be quicker in the worst case scenario than the hash table, but with the use of updates that games have I felt this distinction was redundant. AVL trees are faster overall but with the speed of the search being key in a loot table I felt hash tables were a better suit.

Hash Table

PROS:
Faster Search on average
Doesn't need to be balanced like AVL Tree
Logical structure is followed more than a readable one

CONS:
Slower overall
Fixed size
Less readable

AVL Tree

PROS:
Faster overall
Variable size
More readable
Follows a more human logical structure

CONS:
Requires balancing on each add/remove
Slower search on average

In this program I prioritize the speed of the search specifically so while the AVL tree is tantalizing I feel the hash table serves this purpose better.

Alt Design Sketch

With an AVL tree the program would no longer need the rebalancing structure that the hash table follows and would be replaced with a rotational rebalancing as an AVL tree must be balanced differently. The searching of the tree would also have to change. with the hash table we can hash the numbers even though they aren't the same but with the AVL tree the loot generated would have to be of the same key. This would be the most challenging part as unless all of the key values are kept together it can be difficult to manage what loot should be dropped. An example of how to handle this would be, creating an array with all of the keys and then generating a number from 1-n where n is the total number of keys. Then search the tree for the specified key-value pair. This would disrupt all of the unique items functions as well as the search would be far more time complex then it is with the current system.

Evaluation Plan

The design could be tested in many ways, but the way I would test it is by forcing completion of quests which should have quest loot and defeat of multiple monsters which should also have their own tables. This way I could quickly test to make sure the loot is being dropped at proper rates and by the correct things. I would output some kind of test values to confirm where they came from and how much as well to double check the data. Then after those are all correct, the best way to truly test is to send the game out to beta testers to get a full range of input to confirm that everything is working as planned.

Conclusion

I believe the hash table was the correct choice because of the rate of which the search can be done on average. On average the search function is faster but the main trade off is that, there is a possibility that the operations are slower. This was ok to me because the loot tables shouldn't slow down that much because of the size of the tables.
The other functions that would also be slower could be handled and managed when the player was unaware like during an update or during the booting of the game.
This design is encapsulated from the user as the only thing they can do with it is query the tables and never change any of the values as they aren't public.
The tables are abstracted as well because the features that they will see wont show them the complexity behind the scenes. When they look for the unique items the code and the process are hidden, the only thing they will see is the unique items that are available from the location. In a hash table the user doesn't need to see all of the has codes so with some reformatting into an array, the data can be easily provided to the user.

Citations

“7.1: Time Complexity and Common Uses of Hash Tables.” *Engineering LibreTexts*, Libretexts, 10 Mar. 2021, eng.libretexts.org/Courses/Folsom_Lake_College/CISP_430%3A_Data_Structures_(Aljuboori)/07%3A_Hash_Tables/7.01%3A_Time_complexity_and_common_uses_of_hash_tables.

“AVL Tree Data Structure.” *GeeksforGeeks*, 11 Oct. 2025, www.geeksforgeeks.org/dsa/introduction-to-avl-tree/.