# CS-3100 Project Multiset Design

## Brodey Morrow

### Introduction

###### I will be designing a player inventory that stores the item names and the item count. This will be built upon a HashTable<string, int>. When the player picks up an item the item will have it's name and the amount stored in a HashTable.  This will allow the inventory to keep track of the items already in the inventory while allowing it to change the amount of a given item being stored as well as adding new items or the removal of a type of item altogether.

### Design Philosophy

###### The reason I chose to build the player inventory with a HashTable is because I believe a HashTable is the most efficient way of handling player inventories. It can allow for quick access to the items in the table and the ability to manipulate the amount stored in the hash. The client of the multiset in the game itself and the user is the player of the game.

### Core Operations

######  There are four core operations that a multiset should support. The first operation that the multiset will support is an insert function. It should run at a O(log n) time complexity. When the player picks up an item that item's name is used it's key a hashcode is then generated the item is then stored at the index that corresponds to the hashcode. An

edge case would be receiving an item from an npc. The HashTable is being supported because without the operation there would be no way to insert new items.

#### Pseudo code

```text
DEFINE HashTable AS ARRAY OF BUCKETS called buckets
DEFINE HashTable_size as size_t
Function insert(key,value);
 return true or false
 START
 GET hashcode from key
 CREATE a bucket containing key and value
 IF hashcode matches an index in buckets
 INSERT bucket into buckets
 RETURN true
 IF hashcode doesn't match any index in buckets
 RETURN false
```

###### The next operation would be a erase function. This would run at a time complexity of O(log n). When the player either drop an item or expends an item the item will the index that the item in question was in is cleared. An edge case for this operation is if a item is removed for a quest. The underlying data structure supports the operation because without the operation the table would get clogged with expended items.

```text
FUNCTION erase(key)
RETURN true or false
GET keycode from key
GET hashcode from key
IF keycode matches the key in buckets
RETURN true
IF none of the keys match keycode
RETURN false
```

###### The next operation is a find function. The way this operation would work in game is with a search bar. If the player types the name of an item into the search bar the HashTable is searched for a key that matches the name entered by the player. It should have a time complexity of O(log n). A possible edge case is when the player searches for an item that is not in the player's inventory. The data structure supports the operation because the HashTable is what stores the inventory items.

###### The last operation is a Traverse function. This function would be used to sort the inventory when ever the player prompts it. It should have a time complexity O(n). A possible edge case is if all of the items in the inventory are the same type. The data structure supports the operation because the HashTable is what stores the inventory items.

### Set Operations

###### The set operation that I wanted to cover is the intersection_with() operation. How this could be used in gameplay is having a quest that requires two or more players and those players must have a certain item. This would call the find function in each player inventory and if all players have the item then the quest can be started. A relevant edge case if a player has duplicate items in different hash indexes.

### Extension Feature

###### The feature I would include is a way to assign favorite items. This function would create a new HashTable with the same parameters as the main one. When the player clicks the option to favorite an item that item is added to a HashTable called fav_items. When the player chooses to sort by favorites all of the items in fav_items will be listed. This will help players get to the items they use the most use quicker.

### Trade-off Analysis

###### One of the reasons I chose not to use an AVLTree is that a HashTable offers a much simpler way of inserting and extracting items from a inventory. An AVLTree would requires constant checking of the trees balance every time a node is added or removed and if the tree is not balanced it needs to be rebalanced. A HashTable just requires a key. This makes an AVLTree not ideal for holding an inventory.

**### Alterative Design Sketch**

**###### If I had to choose a different if would be an Sequence. You could insert, erase, traverse, and find by using two nodes the keep track of the head and tail of the sequence. This could work albeit very slowly.**

**### Evaluation Plan**

**###### I would start out by adding every type of item into the HashTable. I would then use the get operation to extract the item from each index in the table. I would then test the remove function by removing items from the table at random. All of these tests should test the time complexity of each operation as well.**

**### Conclusion**

**###### My design allows for efficient insertion, traversal, deletion, and extraction of items. I accept that making sub-items will be harder in a HashTable than in a AVLTree.**

**### Sources**

https://www.geeksforgeeks.org/cpp/multiset-in-cpp-stl/

https://www.sciencedirect.com/science/article/pii/0022000081900337