Name: Hunter McIntosh

12/3/25

Professor James Anderson

Multiset Design Project

Context

This is a design for the use of a HashMap for the containerization of player inventory in Bethesda's next installment of their Elder Scrolls series, Elder Scrolls 6. It will utilize a HashMap to track String key, Int value pairs for player inventory. This multiset will be designed to track the names of items and their quantities that a player has acquired throughout their journeys in the game world. Our users will end up being the programmers who will be utilizing this data structure to create the games UI/UX as well as backend inventory management.

Design Philosophy

The focus of our design will prioritize a few things. Our top priority is the ability to store multiples of items. This is because in an RPG, players will be collecting many duplicates of resources and items. As such a Hashmap was chosen as the data structure that will be used for the core container of this multiset. The reasons for this are that Hashmaps allow us to utilize Key, Value pairs to track item names and quantities.

We can utilize an items name or an internal unique Identifier for the key and then the quantity of the item as the value. This will allow us to track how many of any given item a player has in their inventory at any time. Thanks to object-oriented programming we can also utilize the key as way to keep track of all the items' data utilizing a lookup table that has all the relevant stats and information of that item tied to the given item key. We can do this by utilizing a separate custom data type Class which defines an item as an item object with various stats, descriptions and uses. However, that is outside the scope of this design paper and I only bring it up as another benefit of utilizing a HashMap for our inventory.

Another priority this allows us to check off our list is its quick ability to look up values. By utilizing the keys of a given item, the HashMap will allow us to quickly retrieve that corresponding key's value. In our case this will allow us to quickly find the quantity of any given item based on its name or identifier alone. For example, let's say a player has an item in their inventory with the name "Iron Ore" or an internal identifier of "Ore12345". Since we the quantity is the value of our key value pairs within the map if we look for "Iron ore" within the players inventory we will be able to quickly put on screen the quantity of iron ore a player holds.

The final main priority for why I chose a HashMap over other data structures is that we can very easily and quickly update the values associated with keys inside of our data set. Games have a defined number of items within their system, given that, our keys will stay immutable. However, our values(quantities) will be mutable, and this is where the HashMap helps. Our keys will always stay the same, no matter how many of the given item a player has, its key will never change. Which is helpful for maintaining a speedy HashMap. This will allow our users to, so long as they plan item names/identifiers properly, have near constant time insertions, deletions and lookups while avoiding collisions within the hashmap.

Core Operations

This HashMap based multiset will utilize some core operations. Four of which are going to be Add(Key), Delete(Key), isEmpty() Clear().

Add(key, value=1)

Add will function like insert in a standard hashmap. Provided the key doesn't already exist in our hashmap. It will generate a new Key, value pair of the chosen item key within our set. It will use a hash function to generate the index and set its value to a default of 1 or the provided value. If a collision occurs we will utilize our collision resolution strategy(double hashing) to handle this finding the next best index for the item. This will denote the first new item being added to the players inventory and populate it within the UI. We will then increment the size parameter of the multiset class. Given that we are utilizing a hashmap this insertion method will have an average time complexity of constant time(O(1)) and a worst-case scenario of O(n) should we have collisions the whole way down our HashMap.

Delete(key)

Delete will remove the item from our multiset. In game it will remove it from the character's view in their inventory. Behind the scenes what will happen is we will hash the given key to find the index it resides at. We will then delete the item that exists at that index to free it up for whatever next item the player may find or add to their inventory. We will then decrement the size parameter of the multiset class This will have a time complexity of O(1) on average and a worst case scenario of O(n) should we have to traverse the whole map.

isEmpty()

isEmpty() will return a boolean value depending on if the players inventory is empty or not. How it will do this is it will check the size parameter of our multiset class which will be an integer we have named size which we have been incrementing and decrementing when adding and removing items from the inventory. If size is less than or equal to 0 it will return true. If size is greater than 0 it will return false. This should have a time complexity of O(1) as it is a basic comparison.

Clear()

Clear will delete all the items in the bag. To do this we will traverse the hashmap from beginning to end deleting each index and value as we come to it. In game this wouldn't be something the players would do but would be used by our users(the game devs) for clearing the inventories of various monsters, npcs, chests, etc. should they have a need to. As we are traversing the whole map this will have a time complexity of O(n).

Example:

```
Node newnode = head;

For(int i = 0; i < size; i++){

        Node nextnode = newnode-> next;

        Delete newnode;

        Newnode = next;

}
```

lookUp(key)

This operation will hash the given key to find the index within our map. The value at that index is returned. Collision resolution will be applied as needed. What this will do for us in game is allow our users (the game devs) to quickly and easily assess whether an item already exists within our players inventory or not. As with most operations of a hashmap the look up is on average O(1) and at worst o(n) if we have to traverse the whole map.

Set Operations

An important set operations will be the union_with(otherSet) operation. What it will achieve for us is in most Bethesda games there is a “take all” option for looting various inventories in the game world. What this option does is it takes all of the contents within an inventory and adds them to the players inventory. How this will work will be pretty simple given that we are using a hashmap as the core of our set. The program will go through every index within the otherSet and add the items and their quantities to the players set. If the item exists already they will increase the quantity of that item to be what the current quantity is plus the looted quantity. This will have a time complexity of O(1) assuming smart name choices and utilization of a standard hashmap lookUp operations.

```
Node otherNode = otherSet->head;

For(int i = 0; i < other.size; i++ && !otherSet.isEmpty()){

    If(contains(otherNode->key)){

        Node playerNode = lookup(otherNode-> key);

        playerNode value = otherNodes value + playerNodes value

    } else {

        Add(otherNode)
```

```
            Node tempNode = otherNode -> next;

            Delete otherNode;

            otherNode = tempNode;

        }


}
```

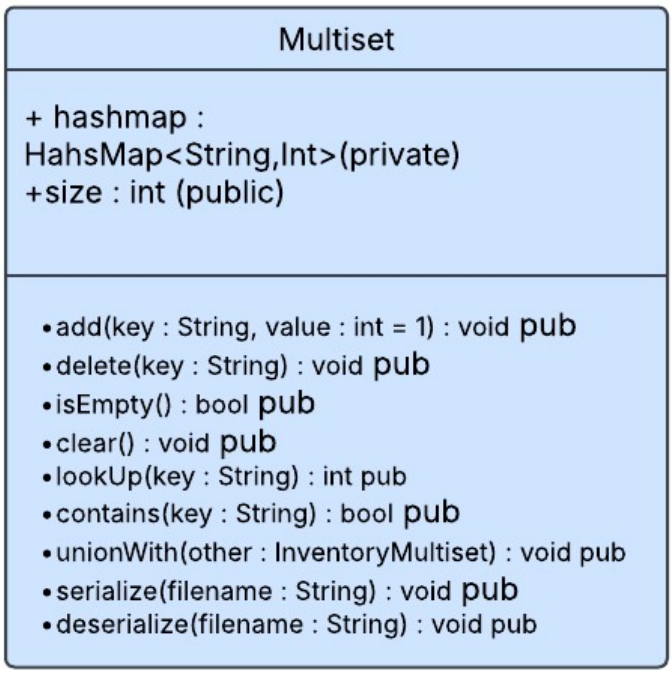
This function will quickly transfer each item within the otherSet into the player's inventory. It will utilize a standard lookUp operation of a hashmap to verify whether an item already exists within the player's inventory or not. As stated previously, assuming our users picked smart item names to be their keys this should operate on average at O(1) for the look up. If the item already exists within our inventory, we perform a basic addition operation to increase the items quantity and then move onto the next item within the other inventory. If an item doesn't exist, we add a new item into our players HashMap and increase it by the prerequisite amount. All in all this function will operate at O(n) as it must traverse through the entirety of the other inventory.

Extension Feature

My new extension feature of my multiset will be a Serialize/deserialize feature. This operation saves and loads the multiset to and from a file. This is important for an RPG. Especially if we want to make sure that our players have local saves. This method will allow us to save our users players inventories at the moment they save so that when they return to the save our users can load the players inventory with what they had at that moment. As such you need

both a serializer and a deserializer. One to save the inventory and one to read the saved data and add all the items back to the player as they load up the save. This will add so much value to our end product as it will give the developers an easy way to save inventories and maintain a defined world state between instances of playing games.

UML Diagram



The hashmap and any functions associated with interacting with it directly are private as they are the background of how our multiset works. Our users will not be interacting with the hashmap but our Multiset. As such they will only have access to the operations and parameters we declare for the multiset itself. This allows us to let our users utilize our bag as a bag while we maintain and handle the internal logic of a hashmap to facilitate the bag.

Alternative design sketch

The trade off of using a Hashmap over a linked list for our inventory management is fairly simple. While our hashmap would struggle to be resized and do so at a very slow time complexity, a linked list is Dynamic. What this means is that our linked list can just allow for quick and easy insertions at the end and just keep growing without needing to be resized. Another huge bonus for the linked list over our Hashmap is it maintains the order in which the

item was inserted. Meaning we can utilize this to do things like remove the last item added or sort the list into another ordered data structure like a stack or a queue.

As such an alternative design approach using a linked list would allow us a more dynamically sized inventory that would hold the order in which players picked items in their inventory. We could also grow or shrink the Linked list so as to allow memory optimization. We could also create operations like pop or push to allow our users to eventually implement stacks with the multiset.

TradeOff analysis

All of this is nice but would not have been helpful for our inventory for one simple fact. The HashMap's ability to look up items and find them quickly and efficiently is going to be the core of our inventory experience for our users and the players. Having the ability to quickly look up items in constant time means that the game is highly efficient in finding and showing all items in an inventory to a player. This will allow us for quick and efficient UI experience for the users players. The indexes are also a huge bonus as this will allow us to quickly and efficiently manipulate our values without having to go hunting for each instance of an item like we would with a linked list. The key is our item and the value is our quantity. If we want to update the quantity of that item it's a simple look up and update to the value.

That is why I chose hashmap over linked list. It allows for quick management of the items and so long as our users implement it smartly should not need to be resized or organized much.

Evalutation plan

To test the design I would first start by testing the limits of the hashmap. Adding in duplicate keys, seeing how the collision resolution would handle receiving a collision once I filled the map, fill a map with known key values pairs and test to make sure the lookup functions work. Then once I verified that lookup and insertion worked I would test deletion. I would also verify that my size parameter worked as expected by adding and deleting multiple items. I would run these tests of functions on 3 different hashmaps, an empty one, a full one and a partially full one.

Conclusion

In conclusion I think for a inventory management multiset like one utilized in common RPGs a hashmap is the best possible choice. Why I feel this way is very simple. The key value pairings allow us to easily keep track of any item by its name or similar internal identifier and also track its corresponding quantity utilizing an int value. This allows for fast look up, insertion and deletion of values. The downsides I've accepted are potential collisions, needing an immutable key, and less dynamic memory when compared to a linked list or similar. Yes the hashmap can still be resized but it does so at a time complexity of O(n) not O(1) which is all a linked list needs to expand its size. If given more time I would have fleshed out how we could utilize another class called Item to store all relevant item stat information to illustrate how our multiset could be

leveraged to allow for using item names to show stats and information in the players inventory as well as quantity.

Works Cited

Olanrewaju, Sule-Balogun. “What Is a Hash Map? Time Complexity and Two Sum Example.” *freeCodeCamp.Org*, freeCodeCamp.org, 25 Jan. 2024, www.freecodecamp.org/news/what-is-a-hash-map/.

“Software Design and Data Structures.” *3.1. Bags - Software Design and Data Structures*, opendsa.cs.vt.edu/ODSA/Books/vt/cs5020/spring-2024/PROD_All_Sections/html/ContentBags1.html. Accessed 7 Dec. 2025.