Jack Thompson

Introduction

My design models a player inventory that stores string item names and counts, built atop a HashTable<string, unsigned int>. The items in the inventory can appear multiple times. My multiset is made to support lots of counting, adding, and removing of elements.

Design Philosophy

My design prioritized efficiency for most used actions like adding, removing, and seeing how many items are in the player's inventory. Having a hashtable will hopefully keep the impact of the basic inventory to an average of O(1). The design will be simple and well documented for other systems to use. The client would be the game and the inventory manager and the users would be the other people developing the game or systems that call any functions that interact with the player's inventory.

Core Operations

The first operation that my multiset will support is adding an item to the inventory (e.g. add(item, amount)). Conceptually it will add or modify an item in the hash table by the amount when the player picks up another of a same item or a new one. The expected time complexity would be O(1) but can be worse with collisions. Some of the edge cases would be if the amount is bigger or would make the amount of that item be more than allowed and that the keys need to be normalized. My underlying data structure supports this because having each item being normalized into a key allows the inventory to look for it much faster than other data structures.

```
bool MultisetInventory::add(string& item, int amount) {
    if (amount == 0) return false;
    string key = normalizeKey(item);
    int current = items[key];
    int newCount = current + amount;
    if (newCount < current) {
        newCount = maxAmount;
    }
    items[key] = newCount;
    return true;
}
```

The second operation would be removing items or item from the inventory (e.g. remove(item, amount)). Conceptually it will do similarly to the add function but subtract from the amount of an item or remove it from the hash table if it reaches 0, when a player uses or drops items. The time complexity would be the same as add and would be expected to be O(1). Some of the edge cases would be trying to remove more of an item than they have. Removing something that isn't in the inventory and removing a negative amount of an item. The underlying data structure would help in the same way as adding an item with having each item being able to be looked up quickly. Another operation would be counting how many of an item a player has in their inventory (e.g. count(item)). It will return what number is associated with the item when the game needs it for checking for like crafting or just displaying it on the UI. The complexity would be a similar O(1). The edge case would be that the normalization would have to be the same as the add and remove operation. Its supported by hashtables by having a very fast time complexity on average. An operation for counting the size of the inventory would be needed (e.g. size()). This operation would count how many different kinds of items the player holds in their inventory. For complexity it would be O(n) as it would need to check each item. It would have an edge case where it could overflow if the count reached too high. I do not think the underlying data structure supports or constrains the operations because other ones would have to go through each as well. Lastly, an operation to get the total count of all items in the player's inventory (e.g. totalCount()). The operation will go through each item in the hash and total the amount of every item combined. The complexity would be O(n) again. It would have the same edge case and neutral support as the size operation.

Set Operations

The unionWith() operation would be very meaningful in my game world. In game play it would allow two inventories to merge, for example the player looting a chest. The operation would take two inventories as parameters. It would take the inventory that is being taken from and for each item inside it would do the add function with the item and amount into the primary inventory. It would have a complexity of O(n) where n is the number of items being merged into the primary inventory. Some edge cases of the operation could be if the primary inventory is full and stops the merge from happening or if the inventory has items that are not allowed to move or in use. The intersectionWith() operation would also be meaningful in the game world. This operation will allow the game world to have quick stacking in chest inventories that moves the same items that are already in the chest out of the players inventory. The operation would take two inventories as parameters. The operation would create a new temporary inventory that has the items that are shared between the inventories. It would go through the smaller inventory and do count lookup on each item in it into the larger inventory. If true it will be added to new inventory. This would make the complexity O(n) as it would only go through one of the inventories completely. The edge case would be if there was an empty inventory making the intersection inventory empty.

Extension Feature

I would add the craftRecipe() capability to my multiset. This would remove the crafting ingredients amounts from the inventory and add a new item as a result if the inventory has all the right items. A new class would have to be made for recipe that would hold items with amount and result item. It would have a support method that would return a bool for if the recipe could be crafted (e.g. canCraft()). The method canCraft() would be valuable in the game world for UI to show the play what is able to be crafted and not. The method canCraft() would take a recipe and inventory as parameters and do count for each item in recipe inside of the inventory. If it returns true for each it returns true, if not it returns false. The method craftRecipe() would take in a recipe and inventory as parameters. It will call canCraft() before running and if it gets false it returns false, if not it runs as normal. After the call, the method will do remove operation on the inventory for the amounts in recipe. Afterwards, it does add for the result in recipe.

UML Diagram

```
MultisetInventory
+ add(item: string, amount: int): bool
+ remove(item: string, amount: int): bool
+ count(item: string): int
+ size(inv: MultisetInventory): int
+ totalCount(inv: MultisetInventory): int
+ unionWith(otherInv: MultisetInventory): bool
+ intersectionWith(otherInv: MultisetInventory): MultisetInventory
+ craftRecipe(recipe: CraftingRecipe): bool
+ canCraft(recipe: CraftingRecipe): bool
- items: HashMap<string, int>
- normKey(item: string): string
```

The normKey method is hidden because it will only be called within the other methods.

Trade Off Analysis

I did not choose to use sequence because of how long the complexity for lookups would be using it. Inventories can get large and it is very easy to implement and scale larger, but inventories require lots of counting items and removing them. With sequence the time complexity would be a lot worse when having to scan through each item every time the inventory need to have items counted or removed.

Type	Advantage	Disadvantage	Average Complexities
HashTable	Maps item and amount, average case is fast	No order and has worst case collisions	O(1)

Type	Advantage	Disadvantage	Average Complexities
Sequence	Fast adding, easy to implement, keeps added order.	Slow lookups and duplicates are stored separately	For adding O(1), others O(n)

Alternative Design Sketch

If I was using sequence I would have a dynamic list of item instances. This would be particularly helpful if the items had a lot of extra data associated with it, not just the amount. For example, having item durability or enchantments like items inside Minecraft. I would implement a separate count variable to help prevent the poor complexities of having to go through the list multiple times.

Evaluation Plan

First, I would have basic tests for all the public methods with different inventory types, like full, empty, single item, and mix. Then I would have tests for the edge cases like trying to use negative numbers or go over the limit of how many items can be stored. I would test the recipe methods and give valid and invalid ones and test for partial crafting. Testing the performance of the design with massive inventories and seeing the latency would be important. After testing is complete, I would attempt to add a new feature to see if the design has good extensibility. I would attempt to have another person look over code and documentation in attempt to see if code is easily comprehensible for others.

Conclusion

Using hashtable to create the multiset inventory is a strong choice over the other options because of its performance and clarity. The design has a very fast key operation complexity with an average of O(1). Which is the most important part of an inventory including adding, removing, and searching for items. The inventory is constantly being updated making it impact the performance of the game heavily. Some tradeoffs from using a hashtable were that the items stored are not ordered in any way, which seemed fine because most operations do not require it, and that hashtables can still have collisions which can affect performance. It shows abstraction by only showing basic operations like add and remove. Encapsulation is shown by having the hashtable and helper methods private and it shows composition by using the outside CraftingRecipe class.

References

Ntarmos, N., Triantafillou, P., & Weikum, G. (2009). Distributed hash sketches: Scalable, efficient, and accurate cardinality estimation for distributed multisets. ACM Transactions on Computer Systems

(TOCS), 27(1), 1-53.

Std::UNORDERED_MAP. cppreference.com. (2025). https://en.cppreference.com/w/cpp/container/unordered_map.html