CS3100 Project 6 - MultiSet

Sidra Ali

The data set I'm intending to use for my game design is Project 4's Hashtable.

1. Introduction:

A multiset is a set that has elements that appear more than once. As such, multisets will serve as a core data structure for the game in any situation where elements appear multiple times. I've chosen the example of the player's inventory.

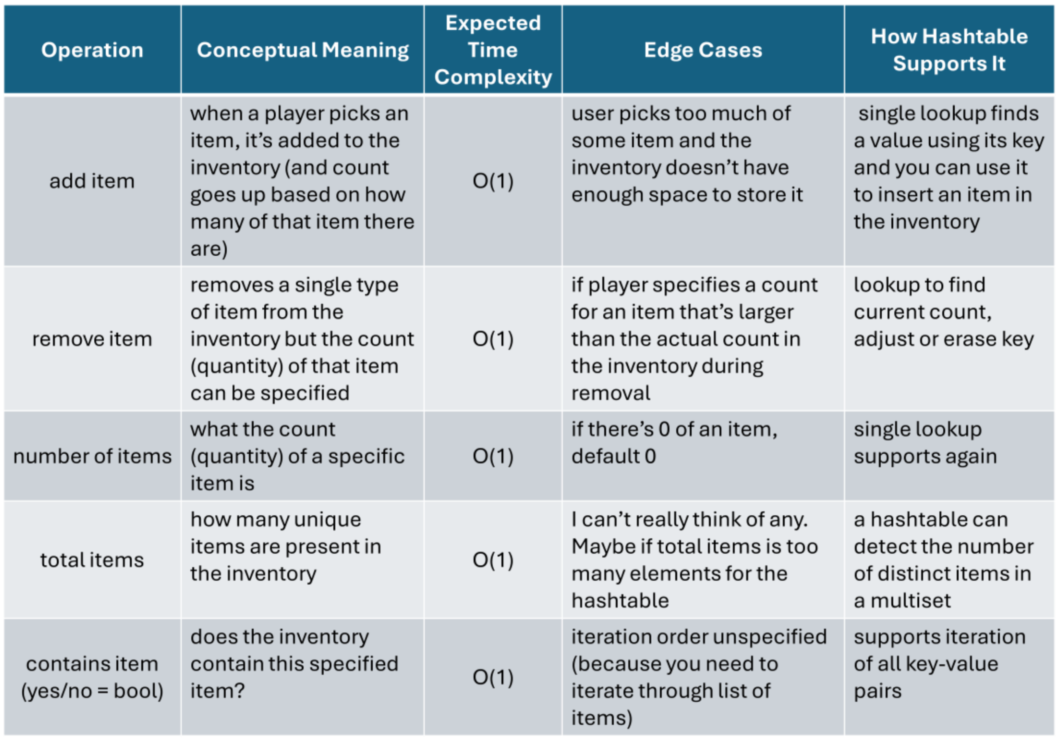
My design models a player inventory for a game where elements are identified by string names and may appear multiple times (e.g. "Potion" × 3, "Arrow" × 12). The MultiSet is built atop a HashTable(HashMap) implementation: HashTable<string, unsigned int> where the value stores the count (quantity of item in inventory) for each string key.

2. Design Philosophy:

Since the player is going to often use the inventory feature, it needs to be **efficient** enough to load quickly and **simple** enough that it doesn't distract from the actual gameplay. I would want the inventory to have a user-friendly interface and make sure it's **accessible** for players of various demographics (like age, fluency in English, people who have difficulty reading a screen, etc.). Another quality the inventory requires is **robustness**, as in it doesn't show any errors or undefined messages if a player does something unconventional. Keeping in mind that I may have future updates or newer versions of my game, the inventory feature needs to be easily **extensible** so I can add features without breaking the pre-existing interface.

The inventory is a core container of the game, but the game will also have other contained systems that may need to work with it (like how Minecraft's inventory needs to work with the crafting table) so the main client will be other game subsystems. Besides that, users of multiset would most likely be game developers.

3. Core Operations:



Operation	Conceptual Meaning	Expected Time Complexity	Edge Cases	How Hashtable Supports It
add item	when a player picks an item, it's added to the inventory (and count goes up based on how many of that item there are)	O(1)	user picks too much of some item and the inventory doesn't have enough space to store it	single lookup finds a value using its key and you can use it to insert an item in the inventory
remove item	removes a single type of item from the inventory but the count (quantity) of that item can be specified	O(1)	if player specifies a count for an item that's larger than the actual count in the inventory during removal	lookup to find current count, adjust or erase key
number of items	what the count (quantity) of a specific item is	O(1)	if there's 0 of an item, default 0	single lookup supports again
total items	how many unique items are present in the inventory	O(1)	I can't really think of any. Maybe if total items is too many elements for the hashtable	a hashtable can detect the number of distinct items in a multiset
contains item (yes/no = bool)	does the inventory contain this specified item?	O(1)	iteration order unspecified (because you need to iterate through list of items)	supports iteration of all key-value pairs

- Table 1: Core Operations
- Source: me on powerpoint

4. Set Operations:

The two set operations I'm going to expand on are `union_with()` and `intersection_with()`

union_with()

- What it accomplishes in gameplay: combines two or more items together. so if I pick up a chest with multiple items in it, I don't have to individually add each item to inventory, they're counted as one thing/union
- How it manipulates my data structure: uses hash lookups on result
- Its conceptual complexity: O(n1 + n2) expected, where n1 and n2 are distinct-key counts/items in the two multisets
- Relevant edge cases: if the order of items being grouped together matters, then you can't use union, you need to add them separately

intersection_with()

- What it accomplishes in gameplay: teo features which share the same item (if I'm crafting a bed and a table, wood is an item that is found in both)
- How it manipulates my data structure: lookup per key in the other table and an insert into result
- Its conceptual complexity: O(k) where k is number of shared keys
- Relevant edge cases: there's no intersection, return null set

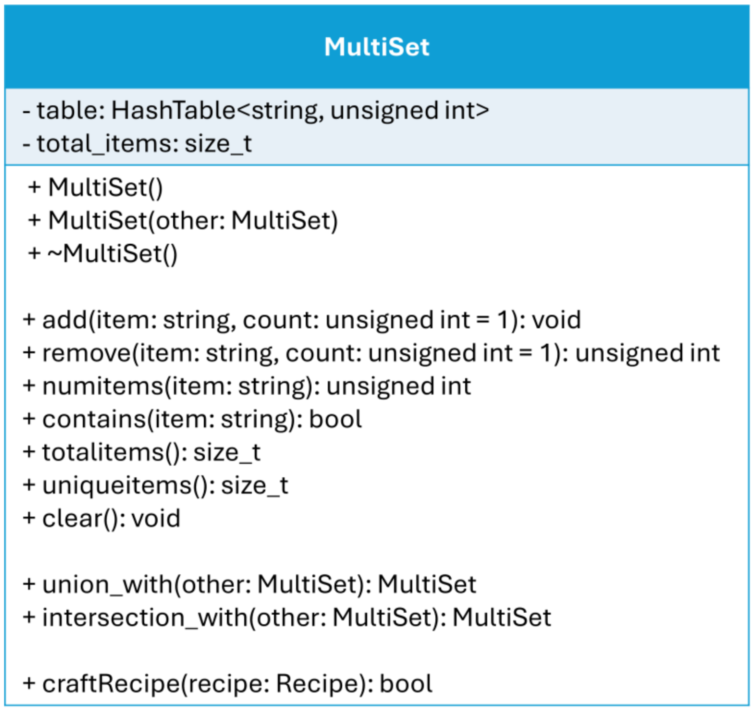
5. Extension Feature:

I'm choosing to add the `craftRecipe()` feature because I'm designing an inventory using multisets. It would be very convenient if items from the inventory can be combined to create other items because when I gain a new item, the intersection can tell me what things I can potentially make from it.

6. UML Diagram:

- is used for private members

- is used for public interface = the player needs to be able to interact with these features



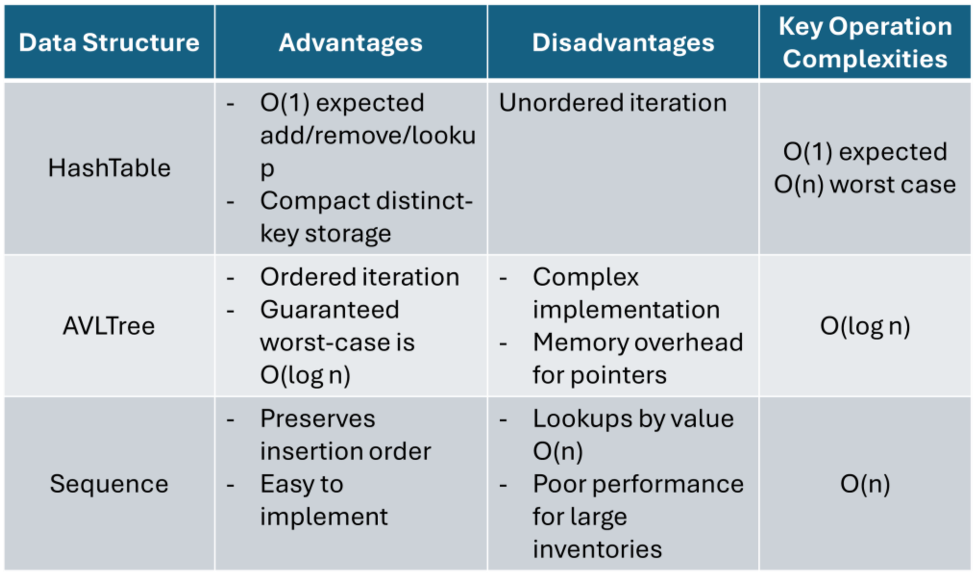
MultiSet

```
- table: HashTable<string, unsigned int>
- total_items: size_t
```

```
+ MultiSet()
+ MultiSet(other: MultiSet)
+ ~MultiSet()

+ add(item: string, count: unsigned int = 1): void
+ remove(item: string, count: unsigned int = 1): unsigned int
+ numitems(item: string): unsigned int
+ contains(item: string): bool
+ totalitems(): size_t
+ uniqueitems(): size_t
+ clear(): void

+ union_with(other: MultiSet): MultiSet
+ intersection_with(other: MultiSet): MultiSet

+ craftRecipe(recipe: Recipe): bool
```

7. Trade-off Analysis:

I chose the HashTable instead of the Sequence because the distinct-key storage and lookup made implementing core operations much easier. Sequence require scanning or pairing key-counts to store counts. It has O(n) lookups which means that if the input size doubles, the time required for the lookup operation will approximately double which makes it inefficient. The larger the inventory gets, the worse a core container based on a sequence would perform. Although Sequence is very easy to implement, it's disadvantages outweight the advantages. Espeically since HashTble has the simplest API (Application Programming Interface).



Data Structure	Advantages	Disadvantages	Key Operation Complexities
HashTable	- O(1) expected add/remove/lookup - Compact distinct-key storage	Unordered iteration	O(1) expected O(n) worst case
AVLTree	- Ordered iteration - Guaranteed worst-case is O(log n)	- Complex implementation - Memory overhead for pointers	O(log n)
Sequence	- Preserves insertion order - Easy to implement	- Lookups by value O(n) - Poor performance for large inventories	O(n)

- Table 2: Comparision of data structures summarized
- Source: me

8. Alternative Design Sketch:

Suppose I did choose Sequence as the data strucutre to design my MultiSet. It could be represented by `vector<pair<string, unsigned int>>` or `Sequence<string>`. The core operations would use O(n) linear search instead of O(1). Whereas, in a HashTable the search would be really simple because of the lookup tables.
Here's some pseudocode showing the difference in how the core operations may look like:

while using HashTable:

```
add(item: string, count: unsigned int = 1) -> void
```

while using Sequence:

```
    if count == 0:
        return

    index = find_index(item)

    if index != -1:
        items[index].value += count
    else:
        items.append( (item, count) )
```

The set operations worst case while using Sequence will become O(n^2). A lot of the automated features of the HashTable would have to be manually programmed, such as handling duplicates of items.

9. Evaluation Plan:

Unit testing is when programmers write test cases for sections of their code. Almost every programming language has its own unit testing framework (e.g., NUnit for C#), which enables the use of small, automatically executable unit tests [1]. I would use this and create my own tests to check:

- the core operations: insert sequences of items and then use the add, remove, contains, and total_items operations and verify the results match what I input.
- the set operations: I'd compose at least two known multisets and try the union and intersection operations with them.
- extra feature - CraftRecipe: if two items successfully combined into a new item, I would check inventory to make sure the two items count went down by 1 in the inventory.

To check performance, I could ask a group of developers/users to try opening/closing/using the inventory at the same time within a timeframe and see how much it slows down the game. I'll time developers when they try to add new features and the smaller the time it takes for the new feature to implement, the better the maintainability and extensibility. Adding new methods should also not require changes to existing clients if the API is stable.

10. Conlusion/Reflection:

I think my design is strong and effective because it takes edge cases (such as zero counts and absent items) into consideration. The interface focuses on the core operations (add, remove, etc.) which makes it intuitive for other developers/game subsystems/clients to use. These core operations all have an expected complexity of O(1) which means that even with the number of users/features increasing, the performance won't suffer a lot. The use of a hashtable makes locating/adding/counting things in the inventory really easy, espeically since it already counts distinct items in multisets.

The trade offs I accepted were the inability to automatically have iteration ordered and that although O(1) is the expected time complexity, the worst case is O(n) which would make performance quite poor. With more time, I could implement a feature that allows for items to be sorted before they are added to the inventory. I could add serialize and serialize as helper methods to store the key-value pairs.

Abstraction: Abstract classes are used to represent general concepts (for example, Shape, Animal), which can be used as base classes for concrete classes (for example, Circle, Dog) [2]. The core operations of the design show abstraction because the add/remove/craft operations can be used in general and don't need to be specified for each item. 'Add' can pick up an apple or a book, without needing separate 'add apple' and 'add book' features.

Encapsulation: Internal HashTable and its members are private. Through the public interface, these private members can be interacted with but the user can't see them/they're hidden.

Composition: The additional CraftRecipe feature showed that the inventory feature can compose with other subsystems in the game.

These principles helped me compartmalize the interface the player is interacting with as completely separate from what's going on behind the scenes.

Citations:

[1] E. Daka and G. Fraser, "A Survey on Unit Testing Practices and Problems," 2014 IEEE 25th International Symposium on Software Reliability Engineering, Naples, Italy, 2014, pp. 201-211, doi: 10.1109/ISSRE.2014.11.

[2] Abstract Class, *cppreference.com*. https://www.en.cppreference.com/w/cpp/language/abstract_class.html (accessed December 2025)