1. Introduction

My design implements a player inventory using a HashTable<string, unsigned Int>. The string would be the key and the item's name. The unsigned int would be the number of those items the player has. The downside to this structure is that the elements are not always in a predictable order [1]. This would make it harder to display the inventory in its entirety in a consistent manner.

IMPORTANT NOTE I refer to stacks in this document many times. This is in reference to a group of items, or one key-value pair. This is not to be confused with the data structure stacks.

2. Design Philosophy

Hash tables ability to access elements in O(1) time is very desirable. I envision the inventory as a grid system, similar to games like Minecraft, The Legend of Zelda: Breath of the Wild, and Terraria. This is the format I prefer and find the most sensical. It lays all items in view for the player and can allow for sorting of items. While this can sometimes make the inventory feel cluttered, it makes everything visible at once instead of having to scroll or search for what you are looking for. The client for this class would be a game that uses an inventory system that allows for duplicates. The user would be someone playing a game that implements that class.

3. Core Operations

1. Display Inventory

- Displays all items in the inventory in a set grid size
- Its expected time complexity is O(n^2) As it will first need to retrieve all keys in the hash table, which will take O(n) time. Then it will have to use those keys to get each element the tables, which will again take O(n) time.
- Possible edge cases include when there are no items in the inventory, and when there are zero of a certain item (which could be good or bad depending on exactly how you want to do the inventory system).
- The Hash table is good for displaying the operation as it's order can be changed easily, where was with a tree or sequence, the order of the items is linked, and in order to change that order, you would need to use another data structure to temporarily store them in and reorder them. The part that holds the Hash table back is that it is slower to retrieve all items since it need to first retrieve all the keys, where a tree and sequence would be able to do the same operation in O(n) time.

2. Search for item

- Searches the inventory for a given item, and returns whether or not the item was found, and how many are in the inventory if it was found
- The expected time complexity is O(n) since it only needs to retrieve the one key
- Possible edge cases include the item not being found, or there being zero of an item, or

multiple entries with the same item. It would need to be worked out if it should only show the item once and combine the amounts, show each 'stack' individually, or something else.
- Hash tables are great for quickly retrieving a data entry, as they can do so in O(n) time. Having the item name be the key will make searching even easier, since the key to be used in the search can be given by the user. The only constraint would be that the user would have to enter the complete, exact name of the item, which could cause the function to feel too strict. There should be a way of creating suggestions when typing or show results that are close to what the user attempted to enter may be necessary.

3. Sort inventory

- This would display the inventory in a sorted manner based off some given criteria such as name, amount, type, latest obtained, last used, etc.
- The expected time complexity for this could vary greatly, and would likely require an additional data structure to store the keys in a sorted order. Just displaying the inventory would take O(n) time. Actually sorting the inventory is multi-step process. For example, even if you were choosing to sort based on number of items of a type, you would still need a way to organize them in the case that there are multiple items in the inventory with the same amount, so you would need to sort the smaller set of values that have the same amount by alphabetially or some other way. This would take O(n^2) time in the worst case. Meanwhile a filter such as last used would never need multiple criteria, as time is linear, and so is the order items were used. Something like that could be done in O(n) time, especially if there was a queue keeping track of items being used. Sorting alphabetically would also require a sorting algorithm, but could use a vector.sort() operation, which takes O(n log(n)) time [2].
- Possible edge cases include there being no items in the inventory, no items having ever been used (or wherever the file containing what items were last used cannot be found).
- Hash tables may not be the best for this since they have no set order. However, the keys can be stored in some other data structure, such as a vector or array, and those keys can be sorted. This vector may need to be loaded into memory at almost all times so that it is ready to quickly pull up all items in the inventory in the same order or be rearranged to be sorted. The only benefit to using a Hash table here is that no matter how sorting the keys happens, accessing the items can be done in O(n) time.

4. Increment Item

- This would increment/decrement and item by a given value. Giving a public method to do this would ensure that any case where the number of items is less than 1 is accounted for and is properly removed from the inventory.
- The expected time complexity is O(1) since Hash tables allow for direct access.
- Possible edge cases include an item being decremented to be less than 1 (in which case they should be removed). Another edge case is when a value is increased beyond what an unsigned int can hold an overflow may occur, so there needs to be a value cap. This cap would likely depend on item type, similar to Minecraft, where some items cannot be 'stacked' while others can be stacked up to 64, and some can be stacked to only 16. Same thing applies when the value would be taken below zero. The number of commands needs to be checked, and possibly return a Boolean value to let whatever called the function know that either know more items can be added/removed.
- A Hash table would work well for this, as it is able to directly access each element and modify its value. There are no real downsides to using a hash table for this function.

5. Add item

- This would add an item to the inventory as either an entirely new entry, or a duplicate of another item with its own value count. The default value would be 1, but it could be specified to be any unsigned int.
- The expected time complexity would be O(1) time, as values can be added directly and aren't linked to other entries. However, since duplicates are allowed there needs to be a check if there is an existing key with the same name, which would take O(n) to check all keys.
- Possible edge cases include the same overflow and underflow type problems as discussed in the Increment Item function. In addition, there is the case where a key is already in the table, since duplicates are allowed, there would need to be a way to distinctly reference each pair with a key, and also have the search function bring up all key-value pairs of that item
- Hash tables are again nice for their ability to add a new key-value pair in O(1) time. However, the item name being the key causes some problems when it comes to duplicates. A number could be added to the end of the key name signifying that it is a distinct value. As discussed in the search function, having a way to return not just the exact match, but also close matches would also return these keys with 1 value added to them.

4. Set Operations

Loot all (union_with())

This function would allow a player to automatically take all items from a chest or other container and add it to their inventory. The proper way to do this is to search if an existing stack (key-value pair) exists and add the items from the container to that stack. If the number of items in the stack exceeds its limit (or rather, if it would), then it should create a new stack and add the remainder to that. It should place that new key in the key vector immediately after the original stack is added to. For items that don't have a matching item already in the inventory they would be added as normal, creating new key-value pairs to the hash table, adding the keys to the end of the keys vector. The complexity would be O(n^2) for the whole operation. As for each item in the container, it will need to check if that item exists in the inventory already, and then either increment a value, or create a new key-value pair. Possible edge cases include the scenario where the inventory is filled. Or when there is a duplicate stack. With my implementation, duplicate stacks would have a number added to the item name to make the key unique. So instead of comparing the keys directly, they would need to compare the strings, and check that the strings at least start with the same characters, and the duplicate stack would be immediately followed by number characters (to ensure cases like **potion of** healing does not get confused with **potion of** swiftness). When the inventory is full, the function should immediately end, and all remaining items should not be added, which is both expected and easy to implement. The code would function something like this:

```
For x in container{
  check inventory for existing item stacks
  while x still has more items && inventory not full{
    add items
    if x not empty yet{
      look for duplicate stacks
      if no duplicate stacks{
        find next open slot to add stack
```

```
    }
  }
 }
}
```

Necessary Items (intersection_with())

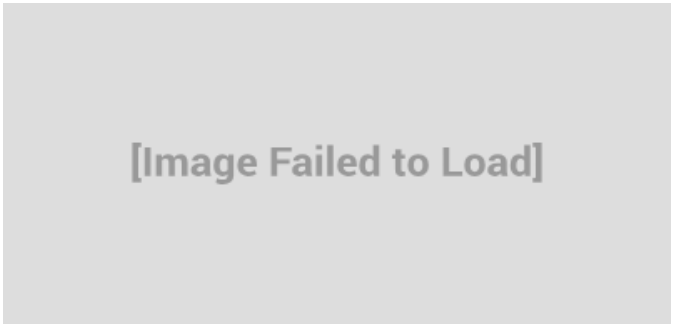
This function would be used to make sure that a player has all the necessary components in their inventory for some action such as crafting, trading, or giving them for a quest. A crafting recipe or trade request would be formatted as new inventory that has all the items and their values needed for that action. It would then check the player's inventory to ensure that all those items and a minimum of that value are present to perform that action. After the check, the transaction would be completed, and it would remove those items from the inventory, using the increment item function, then add the crafted item/traded item or gold to the player's inventory. This would be done with the Add item function. The expected complexity of this function would be O(n^2). This is because it will have to check all items in the crafting recipe/ required for trade, then check the inventory each time to make sure that those required items are there. So essentially it would take O(n) time just to go through the required items, then for each item, another O(n) time searching the inventory, giving a total time complexity of O(n). Possible edge cases include when there is multiple stacks of an item, so even if a stack does not contain enough items, it may need to keep searching to ensure there is not another stack of the required item. This means it needs to be checking the strings of the keys themselves since duplicates will be designated as item1, item2, ... Another edge case is what to do when there are not enough items to complete the transaction. Another edge case is when more stacks of items are received after words than given (i.e., trading 12 carrots for 2 potatoes, a loaf of bread, and a stick of butter). There also needs to be a check to ensure that the inventory will not be overfilled from the transaction before removing/adding items.

5. Extension Feature

Last used items

This function would add to a list of the last used items in order. This can be used for a variety of purposes, including helping them with the sort function to sort by last used items. This would require the class to have a list data field to keep track of the order the items were used. When an item is used, its key would be put into a new head node in the list. Before it does that though, it needs to check if that item is already in the list. Using a queue would be nice, but because we have to check for duplicates at some point, a list will allow for it to be looked through for a duplicate value in O(n). A list is also a better fit than a vector, since it allows for O(1) insertion time at the head node [3]. This is because it only needs to create a new node, set its pointer to the previous head, then update the head node reference. This is better than using a vector, where adding the most used item to the first position would require every element to be shifted up one position. Additionally, it would only need to be a singly linked list, so it will only need to be traversed in one direction (most recently used to used longest ago). This is useful not only for quick item switching to recently used items, but for sorting the inventory by most recently used, as mentioned in the sort inventory function.

6. UML Diagram



The methods and data fields with a '+' before them signifies that they are public, while '-' signifies they are private.

Private members

- **HashTable<string, unsigned int>: inventoryTable**- This data field holds the Hash table with the items and their amounts themselves. This member is kept private because the way the data is modified matters. There needs to be checks that the value does not over or underflow, and there needs to be feedback for when these cases happen so that the program can let user know that the addition or subtraction was not possible. When a value hits zero, that key-value pair needs to be removed. All these checks that need to take place require the data field to remain private.
- **vector orderedKeys**- This vector will hold the order the keys, and by extension the inventory, is in. This way when the inventory is closed and opened it will remain in the same order. This will also allow for the inventory to be sorted and stay in that sorted configuration. This needs to remain private because it would never need to be accessed by anything other than class methods. The user would not need to access this directly, as it would only give item names.
- **list: recentlyUsed**- as discussed in the extension list section, this list would need keep track of most recently used items in order. It should be private as it should only be accessed by another method that would be called when using items. It would also be called by the lastUsed() method.
- **Unsigned int: maxInventorySize**- This data field should be effectively constant to limit the size of the inventory. It should be private as it would only be referenced by methods that add items to the inventory.
- **necessaryItems(HashTable<string, unsigned int>: neededItems):Boolean**- This method should be private as it would only be called by other methods and systems as it serves as a check for other actions to make sure the required items are in the inventory, as well as the correct amount.
- **lastUsed(): list**- This method should be private as it is used to help track of recently used items and would only be called by whatever method signifies and item being used.

Public members

- **displayInventory(): void**- This needs to be public so that the inventory can be displayed to the user. It wouldn't change any data, so there is no way that it could be misused.
- **searchForItem(string: searchItem): Boolean**- This method needs to be public so values can be found in the hash table with risk of modifying the table
- **sortInventory(enum: sortCriteria): void**- This method needs to be public so that a sort filter type can be passed in so that the function knows how to sort the table. While the table would be changed by this function, it would be in a controlled and consistent manner.

- **incrementItem(int: valueToAdd): Boolean**- This method is the public way of adding or subtracting values from a key-value pair. This method has a series of checks to make sure the amount being added or removed does not 'break' the rules set in place or cause other unintended side effects. This method would also allow for table entries to be deleted automatically when their amount is set to 0.
- **addItem(string: key, unsigned int: value): Boolean**- This method is public because it serves as a supervised way of adding new values to the table.
- **lootAll(HashTable<string, unsigned int>: container): Boolean**- This method should be public because it is way of combining and scanning two tables and accesses the data in a safe way.

7. Trade-off analysis

I chose a Hash table mostly because of its ability to directly access elements in O(1) time. A sequence would take O(n) time since it has to start at the beginning every time. An AVL tree would be a bit better with O(log(n)) for element access. Another important consideration for me was sorting, and AVL trees just don't really allow for that. The inventory would always be in a set order. Having a way to keep the order consistent is something that the hashtable can't do on its own but would be even more difficult to implement on an AVL tree. The Hash table does require separate data structure to keep items in a consistent order, which is a point against it. But with that data structure it's as easy as using one key after another to access values. A sequence would have been the best for sorting items and keeping them in a consistent order. Only the reference to the next node would need to be updated in O(1) time when adding or removing an item. However, the sequence brings about another problem, each node only contains a string, so keeping track of the amount of each item would have been troublesome. Overall, the advantage to using a Hash table is that at worst it requires some small work arounds, and at best works perfectly. The other two have high highs, and very low lows. So they may make some implementation details easier, but others much harder. The HashTable made the most sense to me, because it was most similar to how an inventory functions. It takes a key (slot in the inventory) and gives you a value back(number of that item). Its malleable nature made it easy for organization and manipulation. It was the least complicated of the three, which may sound lazy, but it's time complexity for most functions are shorter thanks to its ability to access elements directly.

Capabilities	HashTable	Sequence	AVL Tree
Member Access	Direct access in O(1) time	Sequential access in O(n) time	Binary search access in O(n log(n)) time
Ordering & sorting	Requires a supporting vector to keep track of order and sort elements.	Easy to keep order as pointers can be updated in O(1) time, but requires O(n) to find the value being looked for	Strict order, can't be manipulated easily, would require an entirely separate data structure and take a lot of extra steps to find that order. Might as well just use one of the other two data structures
Addition and removal of elements	Because elements are in order, it again offers O(1) time to create a new key-value pair	Takes O(n) time to add as it needs to find the correct spot to add or remove, but then is able to update pointers to include the new node	Adds or removes element in O(log n). This element cannot necessarily be put in a spot that would make sense due to the AVL tree's requirements.

8. Alternative Design Sketch

The next best choice for implementing the inventory is a sequence, or potentially two sequences as each node on the sequence only contains a string. With just one, it would be difficult to have both an item name and amount. If all operations are performed on both sequences at the same time (or one immediately after the other), then the item names could be stored in one sequence, and the value of that item could be stored in the other. The upside to using sequences is that there would no longer be a need to be a vector to keep an order of items, as the sequences have an order, and that order can be changed by updating each node's pointers. For displaying the inventory, the sequence can be traversed by following the nodes pointers, so the time complexity will be the same. The major difference here is that items would need to be called by their index instead of by their name. This wouldn't change too much other than making it less readable. This may lead to larger time complexities for something like the search function, since instead of using that name as the key to directly access the table value, the program would need search each element in the sequence to find a matching name. This would also make duplicates stacks easier to deal with, as their names could be the same, and the loop could continue until it reaches the end of the sequence to ensure all duplicates are found.

9. Evaluation Plan

With most programs, I try first to ensure that they work for the general use case I had in mind. So first I would add a few items to the inventory with an amount of 1, 2, 3 and the default value. Then I would have it display the inventory to ensure they were added correctly. Then I might increment or decrement them, bring one down to zero to make sure that it is properly removed. I would attempt to sort the inventory with the few items in there. I would create a secondary inventory, then try the lootAll() methods to add all those items to the original inventory, then make a third inventory for checking for required items, and this first go through it would be items the inventory definitely have at this point. After ensuring that it functions at a basic level I would try to break it. I would try to cause every edge case I can think of to see if the error-proofing was effective. I would try to add items with an amount higher than the max, increment to add to make a stack have a higher value than allowed, and attempt to bring the value below zero (which would loop back around because it is unsigned, but it should still check for that). I would try sorting with no items in the inventory, and with a full inventory. I try modifying elements that don't exist in the table with increment. I would test the search function by giving it an exact match, a close match, and an element that has a duplicate stack. I would try to use lootAll() with an empty container, and one with more items than the inventory can hold. Well documented code can help with extensibility and maintainability. This will help others understand how the code works and where to look for certain functions and methods. It makes it so the code can be read like a book so that someone can find exactly what they are looking for. It may also bring attention to a helper functions that would be useful for future additions, reducing the amount of work need to be done, and the efficiency of the code, for future additions/modifications.

10. Reflection

Using a HashTable is a very effective choice for implementing an inventory due to the speed it has for accessing individual elements. The structure of having the name and key be the same helps with searching and sorting, and gives an extra data value to work with each item slot. The

keys being strings make them easy to dissect and uses for advanced searches with close matches. The only real flaw is that there is no set order, and that order must be held in some other data structure. This may slow things down somewhat, but a vector still offers O(1) access for its elements. This makes for a very quick data structure, which is important for quickly getting to items, especially in a more action focused game. This HashTable key vector combo makes for a system that almost mimics how the inventory itself is modeled. With an array of items that can be accessed in sorted, just like the vector of keys can allow for all the elements in the Hash table to be displayed. With more time, I'd like to go more into the sorting algorithm, and how different sort criteria would work, as I find that to be one of the most important functions of an effective inventory. An inventory should be a means of quickly accessing items, not a waste of time fiddling with menus. The class uses encapsulation extensively to ensure that any modifications pass through a series of checks each time. Instead of directly modifying the HashTable methods for adding a new item or changing the amount of an item go through a dedicated method that will ensure an item does not exceed any limits, or wrap back around from going below zero, or if it is exactly zero, automatically removes it from the HashTable. Nothing is modified directly to ensure all rules are upheld and no undocumented behavior happens. These important checks need to happen every time, and it would be a hassle for the user to have to write them out every time. This also falls under abstraction, as a lot of the actual data manipulation would happen within the class, and the user would never have to worry about how exactly it makes sure all data remains consistent. They just have to tell the program what they're adding, how many to add or subtract from an item, etc. If it's not possible, it will let the user know that they are trying to do something that breaks the rules set in place. Composition is covered slightly with the supporting data structures, such as the vector of keys and list of recently used items. A Hash table on its own can't keep track of order like that, but with those two additional data structures added on, it allows for the Hash table to function in a more desirable way, and allow for new capabilities, such as sorting the inventory by most recently used, and then remembering that order next time the displayInventory() method is called. Keeping all the data validation private was important to me, because its less work for anyone implementing the class, and creates a clean and consistent way of making sure that data doesn't break.

References

[1] W. D. Maurer and T. G. Lewis, “Hash table methods,” ACM Computing Surveys, vol. 7, no. 1, pp. 15–15, Mar. 1975. doi:10.1145/356643.356645 [2] cppreference.com, “Std::Sort,” cppreference.com, https://en.cppreference.com/w/cpp/algorithm/sort.html (accessed Dec. 4, 2025). [3] cppreference.com, “STD::List::push_front,” cppreference.com, https://en.cppreference.com/w/cpp/container/list/push_front.html (accessed Dec. 4, 2025).