Project 6 - Multiset Design

Adam Hume
CEG 3100 - Data Structures & Algorithms
Dayton, Ohio
hume.14@wright.edu

I. Introduction

An inventory was designed for a new text-based adventure in the works known as Bork (a pun on the name of Zork). The game has no max capacity for the inventory, so the inventory gets large rather quick. As this is a text-based adventure, items are not called by a numerical index but by a string. In addition, duplicates need to be able to be stored. In essence, a multiset was implemented using an AVL Tree to store the item names as keys, paired with the quantity of each item. This has the added benefit of making it easy to read the inventory in alphabetical order.

II. Design Philosophy

This game needed an inventory system that can hold an item name to be referenced through text, and a number to contain the number of that item the player carries. Additionally, having an inventory that is readable is a huge boon. An AVL tree was chosen to handle the inventory as it aids in printing out items in alphabetical order, which is rather handy as the inventory can get fairly large. As stated in [1] AVL trees are known to have issues with concurrency. While there are solutions, they would require reworking the AVLTree class. Concurrency could make the Inventory work much faster, but the change in speed would not be noticeable by the player. Text based games are not affected adversely by lag (and are not performance heavy on top of that), but having to sift through an extensive unorderly inventory would undoubtedly ruin a player's experience. Readability was prioritized over speed when choosing the AVL tree to be used in the Inventory's data structure, as printing the inventory in an organized format was key. AVLTree allows for great extensibility when it comes to organization, as it maintains itself in alphabetical order. Though an AVL Tree is a lot more complex than some of the other options, its extensibility and readability make it a good choice for a text-based adventure. The primary user of this Inventory is the player, not the developer, so having a readable inventory is vital.

The next thing that must be decided is whether to use inheritance or to simply use the AVL Tree within the class. To allow for a bit of versatility, inheritance won't be used, choosing to simply use an AVLTree object within the Inventory class. This is done to make it so that the data structure used can potentially be swapped out later. It could also perhaps allow for some versatility with Multisets using a potentially different underlying data structure if the underlying methods are similar.

III. Core Operations

There are 4 operations that this design needs to implement a sort of MultiSet: ReadInv, CheckItem, AddItem, and RemoveItem. As an AVLTree cannot store identical keys, duplicates are handled by incrementing the value which is stored with the key, treating it as the quantity of the item stored. This "multiset" differs from an AVLTree as its functions need to keep this key/value relationship in mind when adding and removing items.

readInv()

This function prints out a list of items alongside its quantity in the inventory. By running AVLTree's keys function, a vector of unique items stored by the Inventory can be obtained. One can then iterate through the vector to print out each key alongside it's value. This also has the benefit of displaying each item in alphabetical order by default due to the AVLTree sorting the keys as they are put in. Since one cannot access AVLTree's private methods, the time complexity ends up as O(N log N). The only edge case it to print out "empty" if there are no items so that the player does not get confused when looking at their blank inventory

```
//return keys and values in a vector and loop through to print
readInv() {
    Vector Keys = AVLTree's keys();
    if Keys.size() = 0, print "empty" and return
    for each key {
        print key
        print value
    }
}
```

checkItem(Item)

This function returns the number of a given item within the inventory. AVLTree has a get() function that already handles this, so all the Inventory needs to do with the checkItem function is to simply call get() within checkItem(). As only one branch of the tree needs to be traversed, the time complexity is only O(log N). The only edge case is that 0 is returned if the item is not found in the tree.

addItem(Item, Quantity)

This function adds a quantity of an item to the inventory. AVLTree already has a method that does this task (insert()), except it will simply return false if the key is already in the tree and add nothing. This can be used to one's advantage, however. By using the function within an if statement, we can be use AVLTree's overloaded [ ] operator to directly increment the item's amount by the input quantity. This function can descend the branch up to twice, which would still be a time complexity of O(log N). The aforementioned base case is that if the item is found in the inventory, its number should be increased by the input quantity instead of trying to add a new key to the tree.

removeItem(Item, Quantity)

This function removes a quantity of an item from the inventory. The logic already exists within the AVLTree class for removing an item, but removing a quantity of an item shouldn't remove it from the tree if there are still some of that item left over. This function will begin by running checkItem() as defined above to get the number of an item within the inventory. If the number is 0, the function returns false, otherwise it returns true. If the number is less then the quatity to be removed, AVLTree's remove item function is to run removing the key/value pair from the tree. If the number is greater than the quatity to be removed, AVLTree's operator[ ] overload is used to subtract the quantity from the number of that item. Since the branch can be traversed up to twice, the time complexity would still be O(log N). The base cases are those comparisons listed above.

```
//remove X instances of an item from a player and return false if there
were not enough of that item to fully remove
removeItem(Item, Quantity) {
    Ammount = checkItem(Item)
    if Ammount <= 0 return false
    if Ammount > Quantity, AVLTree[Item] - Quantity
    if Ammount = Quantity, Remove AVLTree's Item and return true
    if Ammount < Quantity, Remove AVLTree's Item and return false
}
```

IV. Set Operations

unionWith(Inventory)

This function is handy for when you defeat an enemy or open a chest taking all the loot. Two inventories need to be combined in either case. This is done by using AVLTree's keys() function then looping through each of the keys in the vector inserting the key (and its pair obtained with an accessor) into the other tree. This would have a time complexity of O(N log N), due to needing to access every Item from an Inventory (N) and needing to descend a branch each time (log N). Luckily, there are no edge cases that are not handled in AddItem already.

```
//Get the data structure's key/value pairs and add to another Inventory
unionWith(Inventory) {
    Vector Keys = AVLTree's keys();
    if Keys.size() = 0, print "empty" and return
    for each Key, {
        Inventory.addItem(Key, checkItem(Key))
    }
}
```

intersectionWith(Inventory)

This function is used to filter an inventory by only returning items shared with another inventory. In game this would be used by the dev to modify loot tables, perhaps using the second inventory to restrict what can show up in the first. It could also be used to implement a maximum inventory size if used with the player's inventory. This is done by using AVLTree's keys() function then looping through each of the keys in the vector and using checkItem() on both inventories. The smaller of the two values overwrites the current Inventory's value for that key. This would have a time complexity of O(N log N), due to the need to access every item of an Inventory (N) and needing to descend a branch twice each time (log N). There are no edge cases.

```
//do a comparison with each key obtained from the data structure, and
return the lesser value
intersectionWith(Inventory) {
    Vector Keys = AVLTree's keys();
    if Keys.size() = 0, print "empty" and return
    for each Key, {
        Shared = MIN(Inventory.CheckItem(Key), This.CheckItem(Key))
        if Shared is 0, removeItem(Key)
        else AVLTree[Key] = Shared
    }
}
```

V. Extension Feature

One last capability is necessary for this MultiSet. It needs a function able to check if one inventory contains every single item at a greater quantity than what the other inventory has. If every item exists at a greater quantity, then the difference between the two inventories should be returned. This would be useful for crafting, quests, and shopping, which all require a check for a list of items to be removed. This function should be run from the subtracting inventory (the list of items being removed) as that inventory belongs to the instance making the check.

useInventory(Inventory)

This function runs the AVLTree’s keys() method to retrieve a list of keys to be iterated through. For each key obtained, it compares its held value with the value of the other inventory using checkItem() to check if that quantity is held as a minimum. If it isn’t the function returns false. But if the other inventory contains every item in the current inventory, then those items are removed from the other inventory and the function returns true. The time complexity would be O(N log N), due to the inorder traversal (N) and needing to descend a branch twice each time (log N). An entire inorder traversal is done to handle the edge case where the subtracting inventory is bigger.

```
//compare each key's value from the inventory, and make sure the other
inventory has that quantity before removing those items
useInventory(Inventory) {
    Vector Keys = AVLTree's keys();
    if Keys.size() = 0, print "empty" and return
    for each Key, {
        if Inventory.CheckItem(Key) < This.CheckItem(Key), return false
    }
    for each Key, {
        removeItem(Key)
    }
    return true;
}
```

VI. UML Diagram / Abstraction Boundary

Inventory
- AVLTree Tree
——————————————————————————-
+ void readInv()
+ size_t checkItem(string: Item)
+ void addItem(string: Item, size_t: Quantity)
+ bool RemoveItem(string: Item, size_t: Quantity)
+ void unionWith(Inventory: Inventory)
+ void intersectionWith(inventory: Inventory)
+ bool useInventory(inventory: Inventory)

Figure 1. UML Diagram

The Diagram above shows the members of the Inventory class. The functions themselves are to only be used by the game's developers, but some of them (like readInv()) are to be called by the player with pieces of text. The AVLTree Object is private as it is only meant to be manipulated through other inventories or by the methods below. This allows for the underlying data structure to be swapped out for another if the need arises. No private methods are needed, as making the AVLTree private means that its public functions are only accessible within the Inventory. In a way, it makes those public methods somewhat private.

VII. Trade-off Analysis

AVLTree wasn't the only possible choice for a data structure to implement a multiset. A HashTable and a Sequence were also considered. The primary issue with the HashTable was that the inventory would shuffle itself whenever it reached a certain threshold, and organizing it each time it was displayed would be very complex to implement. The Sequence had a similar issue as it would just store the values without any numeration, meaning you would have to tally up each item before displaying it to the player. Table 1 below shows the pros and cons of each datastructure

Table 1. Comparison Between 3 Possible Data Structures.

Criteria	Sequence	HashTable	AVLTree
efficiency	Has a time complexity of O(N) for accessing items, and a time complexity of O(N^2) when working with sets: 3rd	Has an avg time complexity of O(1) when accessing items, and a time complexity of O(N) when working with sets: 1st	Has a time complexity of O(log N) when accessing items, but a time complexity of O(N log N) when working with sets: 2nd
simplicity	Yes, it's simple, but working it into a usable inventory is difficult: 3rd	The number/key pairing is simple to work with: tied in 1st	The number/key pairing is simple to work with: tied in 1st
extensibility	Really easy to add to, as there isn't much underlying logic: 1st	The way it shuffles itself when it gets too large my pose issues for some implementations: 3rd	Being organized automatically provides some nice features that can be extended upon: 2nd
readability	One has to tally up every duplicate of an item to get its quantity: 3rd	It shuffles itself as it increases in size making it difficult to read 2nd	Naturally organizes itself making it very readable: 1st

As one can see, HashTable and AVLTree were the best options for the bunch, but the way HashTable shuffles itself when resizing is problematic. Since readability is the first priority, AVLTree was chosen as the data structure to use in implementing the Inventory. While HashTable was faster and had all the same functions, one should not worry about performance and lag too much when running a text-based adventure. The game can afford to be just a little slower if it can save the player time reading through their inventory. Plus, the Hash Table that was being considered does not support concurrency either and is not thread safe. This is because it does not have any extra features to mitigate issues that arise in multithreading with Hash Tables as shown in [2]. As neither data structure can handle multithreading, and speed is not a concern in the first place, AVL Trees are the superior data structure for this inventory.

VIII. Alternative Design Sketch

Had I chosen to implement the Hashtable, not much would have changed. Most if not all the functions used when implementing the AVLTree work with HashTable as well. The most notable difference in implementation is that I would want to limit the inventory size so that the inventory would stop shuffling itself. That, or I would have to run a sorting algorithm on the contents of the HashTable whenever the readInv() function is ran to organize the output.

IX. Evaluation Plan

Naturally an Inventory needs to be tested after being implemented. To do this I would run each function, running readInv() to see how the inventory updates each time. Beyond just checking every function, I would need to check each function's edge cases. That means running readInv() with an empty inventory, running addItem() after the item exists in the inventory, running removeItem() for each of its four edge cases, and running useInventory() with an insufficient inventory. I could also easily swap out the AVLTree with a HashTable and do a bit of testing to see if the two datatypes can be made interchangeable, as that would add some nice versatility to the game's inventory system.

X. Conclusion / Reflection

All in all, Object Composition seems to have worked out just fine for this Inventory. Using an AVLTree object with encapsulation (making the AVLTree member private) should help prevent developers from messing with the Inventory system in a way that would prevent swapping out the underlying AVLTree for something else. The Inventory class is using abstraction to hide the AVLTree so that it can be used as an interchangeable part of a fully functional Multiset. At first abstraction wasn't even considered in this project, but after seeing how it can be used to hide details and make parts of code versatile, it can be said that limiting what is visible can allow for one to see the bigger picture.

References

> [1] Ellis, "Concurrent Search and Insertion in AVL Trees," in IEEE Transactions on Computers, vol. C-29, no. 9, pp. 811-817, Sept. 1980, doi: 10.1109/TC.1980.1675680.
> keywords: {Concurrent access;data bases;parallel processing;performance evaluation;search trees},

> [2] "Optimal strategy to make a C++ hash table, thread safe," Stack Overflow.
> https://stackoverflow.com/questions/7086267/optimal-strategy-to-make-a-c-hash-table-thread-safe