MultiSet Design

Introduction

This document details the design of my MultiSet ADT. A MultiSet is a collection of elements in which elements are able to be repeated (Blizard *Multiset Theory*). My design is for a video game inventory that stores string item names and size_t counts. It will be built on a `AVLTree<string, size_t>`. The string (otherwise called the key) will be the name of the item being affected in the inventory and the size_t (otherwise called the value) will be how many of that item is being affected.

Design Philosophy

My MultiSet prioritizes simplicity as the main quality of its design. I want this ADT to be easy to implement and modify if needed. Game inventories have all kinds of different specifications. Some have weight limits or a fixed maximum number of items, others let you carry as many things as you want with no limitations at all. Because games have all of these variations, I want my MultiSet to be as simplistic as possible so it can be used as a base for any system. Because of this there will be no restrictions on how many key pairs the MultiSet can store.

Core Operations

Some of the standard member operations that should be supported by any MultiSet are Insert, Erase, Size, Find, and Contains (*C and C++ reference*). Here is how I plan to design these five operations.

Insert

The `Insert` operation will take a key and a value then attempt to add the pair to the MultiSet. If the key is not already contained in the MultiSet it will be inserted as a new node using the AVLTree's insert operation. If the pair is already contained in the tree, the value of the attempted insertion will be added to the value of the existing node. This functionality will allow a player to have a dedicated spot for each kind of item in their inventory, and a number to show the quantity of each individual item.

```
insert(key, value) {
    Check if the key pair is already in the tree with AVLTree contains method {
        If it is, set the key pair's value to currentValue + value
    }
    Else, the key is not in the tree {
        use AVLTreeInsert to add the key pair node to the tree
    }
}
```

Erase

The `Erase` operation will take a key and value then if the pair is present in the MultiSet the value will be decreased by the specified amount. If the value of a pair is brought down to zero the node will be deleted entirely using the AVLTree's remove operation. This functionality will allow items to be removed from a player's inventory in a similar way to adding them.

```
erase(key, value) {
    Change the key pair value to currentValue - value
    If the value becomes zero {
        Use AVLTreeRemove to remove the key pair node from the tree
    }
}
```

Size

The `Size` operation will add all the values of each key pair together to find the total number of items contained in the MultiSet. The keys will be obtained with the AVLTree's keys operation and those keys will be fed into the AVLTree's get operation. This functionality will allow the overall number of items in a player's inventory to be easily found.

```
size() {
    make a new size_t called total
    make an string vector called items
    fill the items vector using AVLTreeKeys
        for each key pair in the AVLTree {
            get the value of the key at the current loop itteration index of the items vecto
            add the gotten value to the total
        }
    return the total
}
```

Find

The `Find` operation will search the MultiSet for a specific key and return the value associated with it using the AVLTree's get operation. If the key is found in the tree the key pair's value will be returned, but if it isn't found an exception will be thrown. This functionality will allow the number of specific items in an inventory to be easily found.

```
find(key) {
    make a new size_t called result
    use AVLTreeGet to find the value and set result to that value
    return result size_t
}
```

Contains

The `Contains` operation will search the MultiSet for a specific key using the AVLTree contains operation and return a boolean. If the key was found the boolean will be true and if it wasn't found the boolean will be false. This functionality will allow for checking if an inventory contains a specific item.

```
contains(key) {
    make a new boolean called result
    use AVLTreeContains to check for the key and set result accordingly
    return result boolean
}
```

Set Operations

Set operations are operations that take two MultiSets and compare their contents. The two set operations I plan to implement are `intersectionWith` and `differenceWith`.

Intersection With

The `intersectionWith` operation will take two MultiSets and find common items between them. It will do this by running the AVLTree's keys operation on both inventories and storing the results in two string vectors. Then the contents of set1 will be put into an unordered set so the contents of that unordered set can be checked against set2. If the current element is in both sets it gets added to the shared vector which is returned at the end. This functionality will allow for two players to see what items they both have.

```
intersectionWith(multiSet1, multiSet2) {
    create string vector set1
    create string vector set2
    make the contents of set1 the result of AVLTreeKeys on multiSet1
    make the contents of set2 the result of AVLTreeKeys on multiSet2
    create unordered set mixer and put set1 inside
    create another string vector shared
    for each element of set2 {
        if mixer contains that element {
            add the element to the shared vector
        }
    }
    return shared vector
}
```

Difference With

The `differenceWith` operation will take two MultiSets and find items in the first that are not in the second. It will do this by running the AVLTree's keys

operation on both inventories and storing the results in two string vectors. Then the contents of set2 will be put into an unordered set so the contents of that unordered set can be checked against set1. If the current element is only in set1 it gets added to the unique vector which is returned at the end. This functionality will allow for two players to see what items one has that the other doesn't for situations like trading items.

```
differenceWith(multiSet1, multiSet2) {
    create string vector set1
    create string vector set2
    make the contents of set1 the result of AVLTreeKeys on multiSet1
    make the contents of set2 the result of AVLTreeKeys on multiSet2
    create unordered set divider and put set2 inside
    create another string vector unique
    for each element of set1 {
        if that element is not in set2 {
            add the element to the unique vector
        }
    }
    return unique vector
}
```

Extension Feature

Remove N

The `RemoveN` operation will randomly select a key pair and then decrease its value by a random amount. It will do this by getting all the keys in the MultiSet using the AVLTree's keys operation and then placing them in a vector called items. A string from items will be randomly selected and stored in a single string called dropped. Then the AVLTree's get operation will be used to get the value of dropped and store it in a size_t called range. A random number between 0 and range will then be stored in a size_t called num. Finally, the `Erase` operation will be called using dropped and num. This will cause a random amount of items to be removed from a player inventory. This could be used if a player is defeated, causing them to lose items from their inventory that need to be recovered later.

```
removeN() {
    make a string vector called items
    make a string called dropped
    make a size_t called range
    make a size_t called num
    fill the items vector using AVLTreeKeys
    randomly select one of the keys and set dropped to that key
    use AVLTreeGet to find the value of dropped and store it in range
    randomly generate a number between 0 and range and set num to that number
```

```
    call multiSetErase() using dropped and num
}
```

UML Diagram

MultiSet
-Set: AVLTree -Insert(key: string, value: size_t): void +Erase(key: string, value: size_t): void +Size(): size_t +Find(key: s

This is my UML diagram for the MultiSet class. It has the AVLTree that is working as the base and every operation that has been discussed. I will now go over why I made each member public or private

- Set: AVL Tree

This is private because there is no reason that any player should be able to directly interface with the AVLTree. If they could, they could do anything they wanted to their inventory.

- Insert

This is private for the same reasons as the general tree. If players had free access to insert they could give themselves anything and everything they wanted.

- Erase

I decided to make this public because I can't really see being able to remove your own items at will causing any problems.

- Size

Made this one public because a player should be able to know this. It's one of the main reasons this operation would even exist.

- Find

This one is public because, again, a player should be able to search their own inventory. One of the main reasons this exists is to be used by players.

- Contains

This is public because while I can't think of a SPECIFIC reason someone would use this over find to locate something in their inventory, I also can't see any way having the option could end badly.

- intersectionWith

This is public because I imagine this being used by players during a trading interaction and maybe while looting something. Both of these are something they control so they would need access.

- differenceWith

Just like intersectionWith this would be used during either trading or looting. So it would have to be public.

- removeN

Now this one is private unlike erase because I included it SPECIFICALLY with the idea that it would trigger on a player being defeated by something. Therefore, I can't imagine a situation where a player would manually activate this, so they can't manually activate it.

No AVLTree operations are shown because this isn't the AVLTree UML. They don't belong here so they aren't here. Also, I just thought I could mention that my general throughline to deciding whether something should be private or public is if it allows the MultiSet to be modified then its private. The only exception to that rule is Erase for the reasons said above.

Trade-off Analysis

The other ADT I considered using for this was Sequence. At the time I couldn't picture how an AVLTree could do the job, and I'm still having trouble understanding why you would use a Hashtable. I eventually decided to use the AVLTree after I thought about how much faster it could retrieve data from itself and that the value could be used to track how many copies of an item the inventory contains in a single node. The sequence would have to traverse the entire sequence each time it looked for an item, and it would need to create a completely new node for each copy of an item it would store. The internal operations of the AVLTree are more complicated than those of the Sequence, but I decided that didn't outweigh the benefits. Here is a table comparing the two:

	AVLTree	Sequence
Complexity	More	Less
Speed	Faster	Slower
Insert	O(Log2N)	O(1)
Erase	O(Log2N)	O(1)
Size	O(Log2N)	O(1)
Find	O(Log2N)	O(N)
Contains	O(Log2N)	O(N)

Alternative Design

Now if I DID use a sequence the insert erase and size operations would be very similar because sequence has the same operations that AVLTree uses in those. Find and contains would be a little different because sequence does not have a contains or get operation. Instead, I would traverse the entire sequence using a

for loop that iterates over the size of the sequence and the []operator. I would check each node of the sequence one by one until the desired key is found.

Evaluation Plan

To test my design I would first try adding the same key with value 1 around 25 times and then look to make sure my tree only has one node with the value of 25. Then I would add that key one more time but with the value of 25 and check if the nodes value has changed to 50. Next I would remove 10 of that key and check if the value is now 40. After I would remove the remaining 40 and see if the node was deleted entirely. Then I would try adding actually different keys to see if the nodes are created correctly. I would run size to see if the size is correct. I would run find and contains to see if they give proper output. Then I would make a second MultiSet with half its keys shared with the first and test the set functions. Finally, I would make a separate testing function to call removeN and see if it properly removes a random number.

Conclusion

I think that this design is fairly effective given how fast I had to put it together. With more time I would... probably have programed the entire thing for real instead of just pseudocode. I also would have made the number of nodes and total of the values member variables that get tracked instead of calculating them. The number of nodes isn't even tracked actually, I was going to make that the extension function, but I decided it was too similar to size, and it just felt lazy. I feel the code is effectively abstracted without anything too complicated. I also think it is well encapsulated with what I decided to make public and private. The composition is tricky because I don't really have any member variables to use but the AVLTree code being used here might count?

Anyway that's all I have. Hope its close enough to what you wanted.

References

Blizard, Wayne D. "*Multiset Theory.*" Notre Dame Journal of Formal Logic, vol. 30, no. 1, 1989, pp. 36–66, https://projecteuclid.org/journalArticle/Download?urlid=10.1305/ndjfl/1093634995. Accessed 2 Dec. 2025.

"*C and C++ Reference.*" Cppreference.Com, cppreference, https://cppreference.com/index.html. Accessed 2 Dec. 2025.