INTRODUCTION

My design displays a player's inventory in a game that stores the item names and quantity via an AVLTree<string, unsigned int>. This inventory will list the player's current items by linking the quantity of any given item via its name to the string that will be used as a key for the AVLTree. Additionally, the inventory will allow for more items to be inserted into the tree quickly as the player picks up more items and for items to be deleted once the count of any given item reaches zero.

DESIGN PHILOSOPHY

CORE OPERATIONS

The main operation of the design will be a display function. When the player opens their backpack, this function will display all the keys as well as the quantity of each key to the player. This will have a time complexity of O(n) as every element of the AVLTree will have to be navigated at least once to display the full contents. The only possible exceptional situation for the display function would be if the player's inventory is massive, thus making it difficult to properly display every single item on the screen in an acceptable manner, however, there should be other ways to handle this, such as limiting the amount of items that can be held at one time or displaying the inventory in different pages or tabs. This data structure supports the operation because the elements of the inventory are all being properly stored in the tree and by navigating the tree, each element can be displayed in a specific, consistent sequence (alphabetical or some other sequence if the developer desires).

The next operation of the design would be an "add item" function. This would add a new item to the inventory and thus the tree whenever the player finds or creates an item that is not currently in their inventory via enemy drops, quest rewards, or crafting. This will have a time complexity of O(log n) as it will simply be an insertion method for the AVLTree. There should be no possible edge cases or exceptional situations for this operation as there are no issues with inserting a new item into an AVLTree; it will always have the same time complexity regardless of the size of the inventory. Even if duplicate items were needed, such as another arrow being picked up when there is a full quiver already in the inventory, AVLTree's allow for duplicates, so there would simply be a new node for the next quiver implemented as needed. Due to this as well as the consistent time complexity, the AVLTree supports this operation perfectly as adding new items should cause no issues throughout the implementation.

Next, the operation for deleting an item will be implemented. This function would simply remove the item's key, and thus the node, from the AVLTree when the quantity of the item reaches zero or when the player deletes or sells the item. This would have a time complexity of O(log n) as it would call a remove method of the tree. The only edge case for this function would be if the player tries to delete something that does not exist in the tree; however, this can be worked around with a simple conditional before any possible remove scenario ensuring that no

issues would occur with deleting a node that does not exist. The AVLTree supports this operation because the remove method has such a great time complexity for this operation. The remove method in general for AVLTrees allows for a quick linking of each child node from the removed parent node to the new parent node ensuring the operation performs quickly and without issue.

The final core operation to be discussed will be a sorting feature for the inventory. When the player clicks some kind of sort button, or when the game designers feel it is necessary, the inventory will be sorted based off some feature, most likely alphabetical going A-Z or Z-A and displayed as such via the display operation. This will have a time complexity of O(n) as the entire inventory will need to be navigated to sort the items in forward or reverse order, or some other order if needed by the designers. The main edge case would be if the designer wants to sort the items based off something other than a simple alphabetical order as that would be the key for each node in the tree, making it difficult to navigate it via any other requirement. This could be solved by adding another variable to the key as requested by the developer if needed. The AVLTree design supports this operation by allowing the elements of itself to be organized by the string set up as the key for the node. Due to this, any possible organization requirements for the sorting are already predefined by the balanced AVLTree before they are needed to be sorted.

SET OPERATION

The main set operation that would be used in the game world will be union_with() operation to add a full inventory of items to the player's current inventory. This will be accomplished via large quest rewards, where there is an inventory set into the reward that will automatically combine with the player's to quickly insert the rewards to the player, dungeon/raid rewards, where the guaranteed drop rewards for finishing the task are set into an inventory and automatically combined with the player's, and combining backpacks with another player, where the contents of both automatically converge to show the sum of both for trading or crafting purposes. This will manipulate the data structure in one of two ways: adding a new node to the AVLTree based off of the name of the item and adding the quantity of that item if it is not already in the tree or if a duplication of the item is needed (such as if a max stack of an item is reached during the union) or; adds to the quantity of an item linked to a string key in the tree if that key is already in the tree. This operation would have an O(log n) time complexity for each item being combined via union as, since the tree is already sorted alphabetically by the key, navigating the tree to find if the item is there and the quantity needs to be increased or if the item is not there and needs to be added will always result in log(n) searches of the tree.

EXTENSTION FEATURE

A new capability of the multiset will be a craftRecipe() operation. This operation will allow the user to select a crafting recipe from a menu to "use," or remove a set number of items from their inventory to combine into another item(s) as outlined in the recipe. That newly crafted item will then be immediately put into the player's inventory. Crafting is valuable feature of my game scenario as it allows the player to use their inventory in a more interactive way, turning

their inventory from items that simply exist to be used or sold into elements of a larger crafting structure, with each item potentially being a component of a larger crating recipe that the player is incentivized to work towards. Additionally, it allows for more items to be added to loot tables for dungeons or shop inventories that the player would be interested in acquiring, making those sets of items more relevant in the long term. Finally, it adds more life to the world of the game by showing that every item could have a use, regardless of how important it seems, adding more value to any NPC's in the world that would teach how to craft or craft themselves, and increasing the likelihood of player interaction as there would be more trading between them due to the crafting materials or fully crafted recipes. This operation would mainly add methods that implement both the add and remove operations and methods previously implemented. It would also add data for each crafting recipe to be displayed to the user as they acquire the recipes throughout their journey. It would then check if they have each component of any given recipe before trying to execute that recipe. If needed by the developers, it could also check if they are near or interacting with the required medium to achieve the crafting recipe, such as an anvil or crafting table.

UML DIAGRAM

"PlayerInventory"
- String: itemName - Int: quantity - AVLTree: AVLTree
+ display(): void + addItem(itemName, quantity): void + deleteItem(itemName, quantity): void + sort(): void + craftRecipe(itemName): quantity - printTree(): void - reversePrintTree(): void - insert(itemName, quantity): bool - remove(itemName, quantity): bool - balance(): void - copy(AVLTree): void - clear(AVLTree): void - get(itemName): string

The only elements of this UML that are public for the player to interact with are the functions themselves: display, addItem, deleteItem, sort, and craftRecipe. Every other function as well as the variables are private to ensure that the user can't interact with them in any way. The majority of the private methods solely interact with the AVLTree itself, thus making it risky for the user to be able to access these as that could lead to disastrous consequences. As such, the player has very little choice in what they can interact with. The private methods return either void if nothing needs to be returned, bool to confirm that the method worked properly, or string

to display the name of the item to be used in the crafting method. There are more operations specific to the AVLTree that could be added later as needed for future operations, but these methods should suffice for the current design.

TRADE-OFF ANALYSIS

The other data structure that I decided not to use for this project is a Sequence. The primary reason I did not choose it is because of the time complexity of most of my functions, with a time complexity of O(n) for most of them. Due to the design including multiple items that can be added and removed in any order by the player, it is nearly impossible to be certain that the given operation will be at the head of the sequence (or the tail if it is doubly-linked, like the one we created), thus leading to the O(n) time complexity as the sequence could have to be fully navigated to find and display each element, add an element, and remove an element. Additionally, since the sequence is not orders, sorting the sequence would take a long time as well. While it might seem like both the AVLTree and the sequence have a time complexity of O(n) for sorting, the AVLTree is already sorted alphabetically by its keys, meaning that if the player wants the inventory sorted alphabetically A-Z, which the inventory should be by default since it should be balanced, this particular sorting would be O(1), making it infinitely quicker than sorting the sequence. The main advantages that the sequence would provide would be that adding new elements to the sequence would individually be quicker, as a new node would simply be added to the end without having to do the balancing of the AVLTree and that removing would also be quicker due to the lack of balancing. This could allow for quicker executions of the union_with() operation, assuming that the inventory is not so large that navigating the sequence each time to check for duplicates wouldn't be an issue.
INSERT TABLE HERE

ALTERNATIVE DESIGN SKETCH

If I had chosen to use a sequence, the design would differ in several ways. The adding elements operation would take longer, as it would have a time complexity of O(n) for an insertion method, but it wouldn't have to be sorted/balanced like an AVLTree would meaning that it could be quicker if the inventory is small enough. The remove item operation would be similar as it would have the same time complexity as the adding elements operation, but again, it would be quicker with a small enough inventory due to the lack of balancing. These would both mostly act the same due to them being a connection of nodes linked together, but the differences are distinct. Similarly, the crafting operation would take longer on average with the sequence due to the increased time complexities of adding and removing. The display function would be the same as it would with the AVLTree design, both having a time complexity of O(n), as both need to fully navigate their collection of nodes to properly display their contents to the player. The sorting function would take far more time with the sequence due to the AVLTree already being pre-sorted. If the player or the game wants the items of the inventory sorted alphabetically, either reverse or inverse, then the only action needed with the tree is to display the elements in order or

reverse order respectively. Meanwhile, the sequence must use a sorting algorithm each time it is called, nearly guaranteeing a slower operation than the tree would perform. For any other types of sorting, however, the time complexities of both would be the same. The main increase in time would be with the union_with() operation as that would guarantee the need to have multiple comparisons with the contents of the sequence leading to a time complexity of O(n) for every single check of the merger. For large inventories of either the player, the inventory to be merged via the union, or both, this could lead to a much slower process overall than the O(log n) time complexity of the AVLTree.

EVALUATION PLAN

The primary way to test the design would be to have a set inventory of items, both those that can be stacked infinitely and those that have a stack limit (to test that duplicates work properly) and run through each feature of the design. Since the design uses an AVLTree as an underlying data structure, the adding and removing elements feature should be simple enough to test and should provide the same results as adding and removing from an AVLTree by itself. For the display operation, having a backpack that displays the contents of the inventory always open while testing will suffice to test it. The sorting operations can also be tested on this always-open backpack as needed as well. Having some mock crafting recipes made would be enough to test that the crafting operation works as intended, ensuring that it calls the add and remove operations properly and that it only works when the required conditions are met (sufficient quantity of items, proper location, not in combat, etc.). This test can be repeated as often as required to ensure all scenarios are tested. The union operation can be tested by having other testers consistently perform a merging of their inventories as well as regularly giving out the rewards for quests, dungeons, and raids to the testers to ensure accuracy. Additionally, there could be a simple terminal display of the AVLTree's contents for the developers at all times to make sure that the contents are properly manipulated throughout testing.

To ensure future maintainability of the design, the main function that is required would be changing the contents of each node, either the key or the quantity of each item. For example, if a different type of sorting is required, a new type of key could be added to each node. This would take a bit of work as the tree would probably have to be remade, but by making a proper copy of the tree and copying over each element to a new tree with a new key, this can be accomplished easily. If inventories were to grow significantly larger over the duration of the game's lifecycle, such as a bank that stores several expansions of a player's inventory, the AVLTree should still suffice to properly manage all these elements. Due to the contents of this large inventory being locked behind an in-game bank, or some other form of window that the player must interact with, it is ok if it takes a little bit of time to properly display the large contents as the time for the display should never exceed O(n). Additionally, with the contents of the inventory already being pre-sorted based on the keys of each node, sorting these large inventories should be near instant, only having to display the elements in order or reverse order, showing that the scalability of this design is excellent. Since every function of this design is

based off an AVLTree, as long as the tree is modified properly as needed to adjust to any future changes, there should be no issues with modifying or adapting new features as the game's life continues.

CONCLUSION

My design utilizes an AVLTree design to quickly and effectively manage every function that is required. Due to each item being linked to a specific quantity, adding and removing from those quantities is simple, allowing for an easy implementation of the adding, removing, and crafting functions of the design. Because of this simplicity, adding more complexity to these features in the future should be easy to implement, allowing for creative innovations in the future. Additionally, due to the AVLTree design, performing any required set functions, especially the union function detailed above, should also be easy to implement as comparing the player's inventory or set to another will always have a time complexity of O(log n) due to the contents being sorted by the balanced AVLTree. This sorting also leads to quick execution of the sorting function if the nodes contain a key that is conducive to the sorting style requested. In the future, I would try to work on a way to improve the sorting function so that it could sort on more than just names. While this works in its current iteration, adding more player convenience over time is an excellent goal to strive for. I would also try to find ways to implement more set functions, as the actual implementation of them would be relatively simple due to the AVLTree underlying structure. Finally, I would like to find even more creative ways of manipulating the player's inventory as they request in the future as I know that these current functions are basic and players would love to have more easily managed complexity with their game.

This design shows abstraction by hiding the full details of the AVLTree behind the scenes of the player's inventory. The player will only ever interact with their own inventory systems; they will never interact with the AVLTree itself. Each time they call a function of the design, the underlying methods of the tree, add, remove, sort, etc., will be performed without those methods interrupting the user experience of playing the game. The design exemplifies encapsulation by bundling all the internal data, the nodes with their keys and quantities, in private data fields inside of the class. The methods that interact with the AVLTree will also be private, meaning that the user can never directly interact with the contents of the tree itself ensuring safety. The player can only ever interact with the public methods, being the operations that call upon the private methods to perform each function, ensuring that no data manipulation can occur. Finally, this design shows composition by having every function build off the basic methods of the AVLTree. Each function uses the methods of an AVLTree, whether those be search, add, remove, copy, delete, balance, etc., as an underlying basis for the function itself. For example, the crafting function calls upon the search function to see if the components are in the player's inventory, the remove method to consume the ingredients, the add function to add the result into the inventory, and the balance function to balance the AVLTree if needed after the removal and insertion. Overall, these principles helped my decision-making during this project by allowing me to use what I've learned throughout this class, build upon those principles that I've learned, and turn

that knowledge gained into a simple, cohesive design document that honestly did not take much time to fully come up with. Fully understanding these principles allowed for an easy implementation of this design and will assuredly lead to further simple designs in the future.