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

TRADE-OFF ANALYSIS

ALTERNATIVE DESIGN SKETCH

EVALUATION PLAN

CONCLUSION