# Introduction

My design will model a player's upgrades, inventory, and boosts. It will be able to manage the three different types of upgrades the player can continue stacking, the inventory for the three different boost bottles the player has collected but not used, and the boosts the player has used which can also be stacked up. This will be built using Sequence<string>.

# Design Philosophy

A quality that was prioritized when considering the function of the game was, first, extensibility. I wanted to make sure the game could work infinitely both within the game itself and as a set of code in the real world. "While many representations might seem plausible, we must select one in which the location in memory can be determined efficiently" [1]. Simplicity was the next quality to look for because it would be easier to read and to edit in the future. Finally, it needed to be an enjoyable game for a mobile device played by nerdy gamers.

# Core Operations

*Insert()*. Being able to add items to a stack is extremely important for almost any game or code. This is the key concept of my game, being able to stack or "collect" boosts and upgrades. This is simple to do in a sequence and has an expected time complexity of O(1). If there was ever an additional item, an overflow of items, or any reason for a new array slot, there would be increased time complexity (O(n)) and possible game miscalculations. The sequence data structure supports this by the storage being handled automatically, being expanded as needed [2].

*Count()*. Being able to read all the items currently in use will be a part of my game design. This will be used when a player uses multiple of the same boost to increase the time it will stay active.

This will be done by reading down the list of each boost type and adding time for each one in the list. If it is called on an empty slot it should return zero. The time complexity of this is O(1) with basically no exceptional situations because the underlying data structure was intended to be read like this.

*Remove()*. Boosts will not last forever and need to be deleted. Also, when a boost bottle is used from the inventory it will need to be deleted. When deleting items I will need to make sure the boost time-remaining is not messed up, that random items do not get removed, or that items get deleted from the inventory before they are accounted for. Also, if it is called when there is nothing to remove, it should do nothing. This will have a time complexity of O(1) since we are only ever taking items off the end of each list which is supportive to our data structure.

*Clear()*. Being able to remove all the boosts the player has or clear the inventory is a typical part of every game. Having the ability to clear one to three boosts from a negative side-effect, clearing a level, or also clearing the inventory upon death will great in my game. This is the slowest operation with time complexity of O(n) as each element of the dynamic list has to be cleared individually. However, it is fast to find what to clear since it is ordered in our data structure.

## Set Operations

*intersection_with()*. A possible relationship between game inventories is to have two inventories combine the shared items from each other. This would allow the player to loot containers with various items in them quicker and progress into the game more seamlessly. Each container in the game would have the same as the player but with only the inventory slots available or visible. Then this operation would take each item from the list in the container and add it to the player's

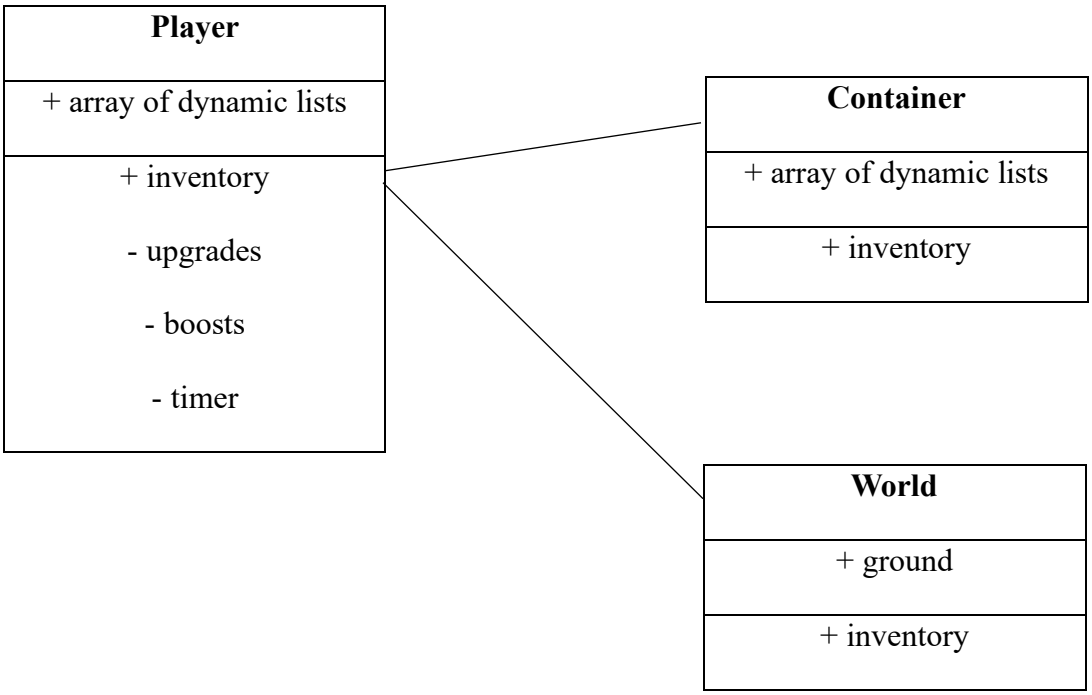
inventory item slots. This has a time complexity of O(n) where n is how large the container is and as long as it does not overflow there should not be any edge cases (except only one item in the container is O(1)). This could be implemented by:

```
intersection_with(player, container){

while(container.size() > 0) {

move all of slot 3 into player’s slot 3, then

move all of slot 4 into player's slot 4, then

move all of slot 5 into player's slot 5.

}}
```

## Extension Feature

*use_all()*. Another capability of the player will be to use one of all boosts in the inventory at the same time. This would check each slot of the player’s inventory and if it is not empty then a boost bottle will be removed and applied to the player. This is another way for the player to save time with inventory buttons, and you will be able to focus on game progression instead.

## UML Diagram




Upgrades are only available for the player to directly upgrade and should not be touched otherwise, so they are private. Boosts are also only available for the player to deploy and all other game functions for it are automatically done for it. The Timer is to remove a boost module every time it ends. Thus, neither of those need to be public.

## Trade-Off Analysis

| Sequence | AVL Tree |
|---|---|
| + Index Access | - No Index Access |
| + Best At Deleting At End | - Not Good At Deleting At End |
| + Best At Inserting At End | - Not Good At Inserting At End |
| + Only Sort Once | - Sort Multiple Times |
| + Grows Dynamically | - Grows Systematically |

I chose to use a sequence over an AVL tree because a tree is much harder to store an inventory of items in a meaningful, clear way. Detecting how many of a boost is in use and where to add new ones, especially “infinitely” while collecting items and constantly rebalancing the tree would be cumbersome and become disorganized quickly. Then having those boosts and items being removed constantly at the same time would require many more calculations than a sequence.

## Alternative Design Sketch

An alternative design for this game could be created with a HashMap. It would share the same time complexity for a lot of operations, except for a few that are worse. Otherwise, it would still have a similar layout and plan as before, but it would be implemented into a HashMap instead of a Sequence.

## Evaluation Plan

There would be multiple tests done throughout the implementation of this design. After every few new lines of code, a test would be done to make sure it compiles correctly. There would be set input variables to continuously test until I got the desired results at first. I would see what is going on during and after the code is complete by printing statements to the console. At the end I would be using random inputs to test all possible outcomes and would have no temporary outputs remaining (to the console).

At this point I would need to clean up my code completely of temporary outputs and commented out code. I would then need to make sure it is properly formatted and spaced out to be clean and

legible. Finally, I would go through the code and comment on anything that could be unclear like what a block of code does or where a line of code goes too. This should allow the code to be maintained for much longer and can then be extended and expanded upon more easily in the future.

## Conclusion

A sequence would be the best data structure for my game idea. It would allow me to create an inventory system, upgrade system, and boost system easily. It would allow me to access other containers to pull out all the items and use all the available boosts at the same time. If there was more time to improve there would be more boost bottles, reasons to remove boosts, and a more detailed timer system for them. For example, the timers are private because nothing should change them once they are going, which will create skill and tension when adding boosts to the player over time. The player is simplified into one composite array of dynamic lists which each has their own meaning. The first three are for upgrades, the next three are for boost bottles (aka the inventory), and the last three are for the active boosts on the player. Then we can use the same positioning/indexing logic to keep containers inventories in the same place for easier transferring of items. This level of encapsulation, composition, and abstraction is what helped me design my game from the beginning through the end.

# Citations

[1] E. Horowitz and S. Sahni, “Fundamentals of Data Structures.” Available: https://ggnindia.dronacharya.info/Downloads/Sub-info/PPT/2ndYear/ADVANCE-DATA-STRUCTURE-Book-2.pdf

[2] “std::vector - cppreference.com,” *Cppreference.com*, 2025. https://cppreference.com/w/cpp/container/vector.html (accessed Dec. 05, 2025).