

# AVLTree Loot Table

Carson Rueter

December 5, 2025

## 1 Introduction

Games that contain enemies to defeat or chests/treasures need a way to determine what to give the player and when. Typically, this is done randomly (or pseudo-randomly, more accurately). However, if the loot was totally random, rare and useful items would be just as common as lesser useful items in the list of loot. To solve this, game systems will often implement loot tables; these assign "likelihoods" or probabilities to specific items within an enemy or chest's list of drops. Additionally, they must decide on the *quantity* of items that get dropped; for example, a small mechanical enemy may only drop 1-3 iron or steel, but a larger, upgraded mechanical monster may instead drop 2-9 iron or steel.

My design models this loot table as an AVL tree. The tree's keys would be string item names (or integer IDs, depending on the game engine), and the values would be integers. These integers would both represent the *maximum* amount of items that can be dropped, but by extension also the *probability* of these items getting dropped. The rolling is up to the client, but an example may be that a mechanical monster has the value of "Iron" at 20 and "Gold" at 4. This means it can drop a maximum of 20 iron, and would drop 10 on average, compared to a maximum gold drop of 4 with an average of 2.

I primarily chose this design as it enables quick lookups, additions, and traversals, which are essential for a structure that is used quite frequently in combat RPGs and survival games.

## 2 Design Philosophy

My design prioritizes efficiency, determinism, and extensibility.

Determinism is important in this case because it ensures that running operations on this table are always done with the same time complexity. Efficiency is important because this structure will be used frequently; every time any loot is calculated, such as when the player opens a chest or kills an enemy.

Extensibility is important as it allows for easy modification to this loot table. For example, if the player has a sword with an enchantment that increases the drop rate of iron, the game engine can simply multiply the drop rate of iron in a given enemy's loot table by some number. If the game engine wants to only allow for single drops and use the value as a "probability" in and of itself, that can be done as well.

The *client* of the MultiSet class is the game engine—specifically any kind of loot drops, such as from a treasure chest, quest reward, or felled enemy. The *user* of the MultiSet is the player or players who interact with those loot drops in any way.

### 3 Core Operations

A MultiSet should support the following core capabilities, in terms of a loot table:

#### **add(item, weight)**

Increases the value of an item being awarded (and thus the weight and maximum). This would primarily be used in cases where the user has a "lucky charm" to increase chest rewards, a "loot plus" enchantment on a sword to increase enemy drops, or a "good Samaritan" status that increases the rewards they get from quests.

Implementation:

- $O(\log n)$
- Edge cases: If the item does not exist, a new node is created with the specified weight. If the input is invalid (e.g. a negative weight), then it's rejected and not added to the tree. This *may* be changed by the implementor in case they wish to use some sort of special behavior for a negative weight.
- AVL Trees allow for quick insertion of new items and quick traversal for existing items. This also means that a scenario where the key already exists and we need to find where to add the new value is relatively quick as well.

#### **decrease(item, weight)**

Decreases the value of an item being awarded (and thus the weight and maximum). This would primarily be used in the opposite cases of the add operation:

- The player has a curse that decreases specific rare item drops or increases "detrimental" drops.
- The player's weapon has a drawback on rare loot, e.g. an otherwise "overpowered" sword.
- The player has an item that decreases common loot drops but increases rare loot drops.
- Alternatively, a game engine may choose to decrement the weight of an item in the table every time it drops. This would result in cases of getting a ton of items at once being uncommon (for balancing), though it may require extra logic to not reduce the maximum drop count depending on the exact implementation.

Implementation:

- $O(\log n)$
- Edge cases: if the item does not exist, nothing is done. If the weight is unspecified or is equal to the item's current weight, it is removed entirely. An empty tree is a NOP.
- The benefits of an AVLTree for this operation is the same as add; traversal, searching, and removing are all very quick operations even for large trees, and we don't have to handle the horror of duplicate keys.

## **weight(item)**

Gets the "weight" of an item in the tree. This would be used for calculating drop rates when needed, such as opening a chest or defeating an enemy, or for potentially displaying this information to a user or a documentation engine.

Implementation:

- $O(\log n)$
- Edge cases: if the item isn't in the tree, just return 0 and the game engine must know this means it's not a valid drop. Alternatively, it may return -1.
- Using an AVLTree for this is ideal since tree searching is very fast.

## **contains(item)**

Check if an item is contained within the loot table. This is useful if the game engine maintains an "encyclopedia" of loot drops; for example, the user can check their encyclopedia for Ivory, and the game will search through enemy/chest drops using this function and let the player know that elephants and certain treasure chests may drop Ivory.

Implementation:

- $O(\log n)$
- Edge cases: If the tree is empty or the item isn't in the tree, it will return false.
- Like the weight function, AVL trees have a fast search method, meaning this operation won't take too much time.

## **items()**

Returns a list of all items contained within the loot table. This is useful for game engines that choose to only show the *list* of items contained within an enemy or chest's loot table, but require e.g. a separate power-up to see the weights.

Implementation:

- $O(n)$ , but a client could trivially maintain a vector of items that are contained within the tree to make it  $O(1)$ .

- Edge cases: empty = return a blank vector.
- The AVLTree does not require multiple traversal to get to all of the items, so this operation is relatively fast.

## 4 Set Operations

### `union_with(other)`

The union operator would combine two loot tables together, summing together shared drops and adding any unique items to the table directly. This has plenty of uses within gameplay:

- A system that looks through the loot tables of enemies in the area, and does some math to determine what items are most commonly found in the area (e.g. This area has lots of Steel!)
- A game that has an "enemy fusion" gimmick can merge their loot tables together to get the total loot drops of the combined grotesque specimen.
- A special type of weapon that "adds" certain drops to loot tables can simply call this method rather than individually adding its items (e.g. a sword that makes zombies able to drop gold and potatoes).

This manipulates the AVL Tree with the previously-defined add operator. Since that operation already accounts for "duplicate" keys and "new" keys, no extra logic is needed here to handle that.

Algorithmically, you are running an  $O(\log n)$  operation  $m$  times, one for each time in the other tree (with its length defined as  $m$ ), thus making the time complexity  $O(m)$ .

Edge cases:

- One tree is empty = NOP if the other tree is empty, or a deep copy if the source tree is empty.
- No items overlap = still an  $O(m)$  operation, no real problems since add already handles this for us.
- Ditto for all items overlapping.

Pseudocode could look like:

```
LootTable union_with(LootTable other) {
    for item in other.items {
        this.add(other.weight(item))
    }

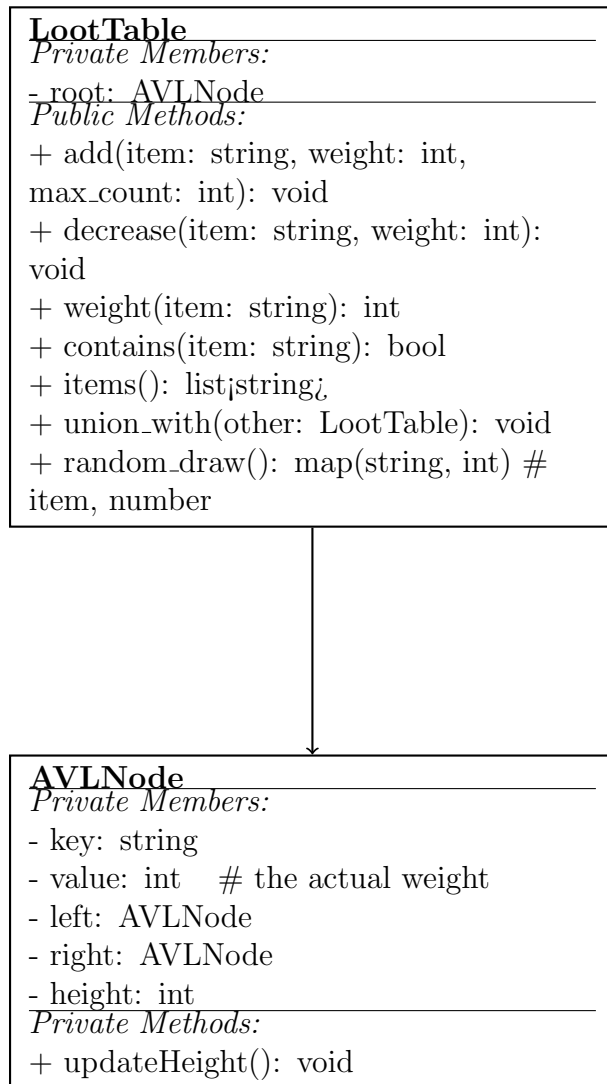
    return this
}
```

## 5 Extension Feature: Random Draw

To serve more as an "engine" rather than a basic structure, the MultiSet could also define the actual draw operation. This would randomly draw items from the loot table based on their weights (= max count and used to determine probability). This would provide a "sane default" method of drawing items from the table, meaning that developers who just want the default behavior and a basic "tell me what to drop!" don't need to implement it themselves. No new data or methods would need to be added; we can simply iterate over the output of `items()` and call `weight()` on each.

The "probability" of an item dropping could be calculated in many ways. For example, we could sum up all the weights and use  $\text{item weight} / \text{total weight}$  as the probability (this may necessitate an additional total weight function), and then roll that probability (item weight) times.

## 6 UML Diagram



## 7 Trade-off Analysis

Table 1 compares the AVLTree to other structures.

Table 1: Trade-Off Comparison of Loot Table Structures

Structure	Advantages	Disadvantages
AVLTree	Sorted order, fast operations	More complex implementation
HashTable	Generally fast lookup	No ordering at all, can degenerate into having a terrible worst-case
Sequence	Trivial to implement	Slow search and removal (horrible worst-case), or requires extra expensive logic to keep it in a good form.

Since the AVLTree is always deterministic and is generally the fastest, this is what I went with. If I instead went with a Sequence, I would have to implement lots of extra weird logic that would not suit a "map" like this well. Sequences only really define a "list" of objects, not a map. Because of this, implementing a key-value system for this is woefully inefficient, as most (common) operations would be in  $O(n)$  time. Additionally, merging two loot tables would be woefully inefficient, due to all of the nested linear searches that would be necessary to do this operation.

## 8 Alternative Design Sketch

A HashTable implementation would have somewhat similar operation performance on average. In the best-case scenario, a HashTable would indeed be significantly faster than anything an AVL tree could do. However, depending on the hash function, item names, and number of buckets, the HashTable would either have a limited number of slots in the loot table, potentially degenerate into a linked list, or be non-deterministic in its ordering and operations. This would be a problem for UI listings and be significantly harder to debug. With the AVLTree, although some operations may occasionally be slower, it's dramatically more consistent and deterministic, so it's more suitable for a "general" application like this.

(A sequence would be so inefficient it's not even worth considering for anything at any remote scale.)

## 9 Evaluation Plan

Testing would include:

- Unit tests for edge cases, such as with empty trees, complex unions, etc.
- Stress testing with thousands of items; searches, additions, removals, etc.

- Randomized tests to verify that the random draw function has approximately expected results.
- Integration tests with combat and chest generation code.

Extensibility and maintainability could be tested by seeing how easy it is to implement or modify features of the AVL Tree itself. For example, if a client wants to change the way the weight distribution works, how easy is it for them to do that? If we wanted to change the internal implementation of union to only *combine* common items, how easy is it for us to do that?

## 10 Conclusion

Using an AVL tree creates an effective loot table due to its deterministic behavior and fast operations, which is important due to how frequently a loot table will likely need to be looked at in, say, an RPG. The internal implementation is far more complex than a hash table or sequence, but the efficiency and determinism are far more important for the end user than internal implementation burden.

Demonstrations:

- abstraction (exposed public methods): This design is not an actual implementation, but a definition of everything needed to make the implementation happen. This helped shape the project by forcing me to think at a *high-level*, specifically how the USER will interact with it. This is definitely an important skill to learn, because focusing on the internals of libraries or reusable components above all else has certainly caused me problems in the past (though it can be useful at times, like for UI libraries).
- encapsulation (private functions/details): The end user doesn't really need to know the internal structuring of the AVL tree, so we "hide" internal details like the node objects, private helper functions and member variables, etc.
- composition (Nodes): The LootTable class contains AVL nodes, but is *not* a node itself (is-a vs. has-a). Since the LootTable (well, and the nodes themselves) contain nodes, they themselves don't have to worry about the internals of the node, and don't have to deal with weird inheritance shenanigans.

## References

- Adelson-Velsky, G.M. & Landis, E.M. (1962). An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*.
- C++ Reference Documentation. Retrieved from <https://en.cppreference.com/>