Project 6 - MultiSet Design

Data Structure and Algorithm

December 6, 2025

1 Introduction

Based on what I understand from description of project 6, I had two options: an RPG game or a tower defense game. An RPG game is about a hero's journey, while Tower Defense is about strategic defense. I decided to go with the "Tower Defense" game. I like this kind of game more. This design will propose a class named "DefenseTower", a multiset data structure for core logic of Tower Defense game system. The player will manage defense strategies and structures to stop the incoming enemies. As mentioned in the project pdf, multiset (DefenseTower) will support multiple instances of same item e.g. a player likes to have 5 cannons and 2 towers to attack. As mentioned in Table 1, the comparison between Hash Table and AVL Tree bring me to the decision of choosing Hash Table (HashMap)($< string, unsignedint >$) as shown in Figure 1.

Comparison Criteria	Hash Table	AVL Tree
Search Time Complexity	O(1)	O(log n)
Insertion Time Complexity	O(1)	O(log n)
Deletion Time Complexity	O(1)	O(log n)
Memory Overhead	High	Low
Range Searches	Requires special implementation	Efficient
Re-balancing	Not necessary	Required
Recursion	Not Inherently RS	RS
Implementation	Mostly relies on Libraries	Easily Customizable
Suitability for Small Data Sets	Less suitable due to memory overhead	More suitable

Table 1: Comparison of HahsTable vs AVL-Tree[Gee]

Hash Table for game design is perfect because it will check the inventory (game loop Figure 2) thousands of times in a simple game. A Hash Table is the only structure fast enough to do this instantly (O(1)), so the game doesn't lag/stutter.

2 Design Philosophy

There are three primary qualities in designing the DefenseTower: ***Efficiency***, ***Simplicity***, and ***Extensibility***. But before talking about them, let us define the latency, and compaction in the operation system.

Latency is a measurement of delay in a system.Network latency is the amount of time it takes for data to travel from one point to another across a network. The higher the latency, the slower the response times[IBM]. Compaction is a technique to collect all the free memory present in the form of fragments into one large chunk of free memory, which can be used to run other processes. It does that by moving all the processes towards one end of the memory and all the available free space towards the other end of the memory so that it becomes contiguous[Gee24].

2.1 Efficiency:

I believe in a game, latency (response time) is more important than memory compactness. In a tower defense game, operations like checking if a tower is available (contains) or removing a tower (remove) occur frequently.

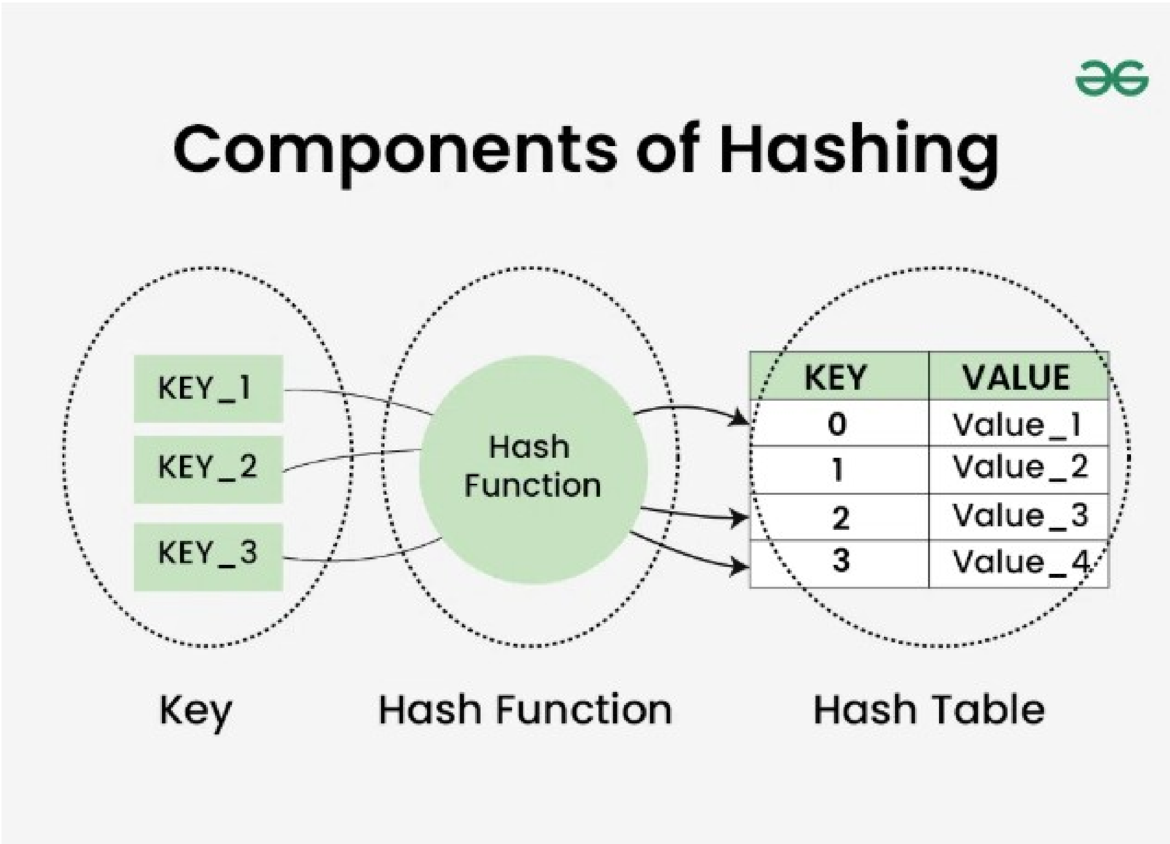

Figure 1: HashTable[VCF16]

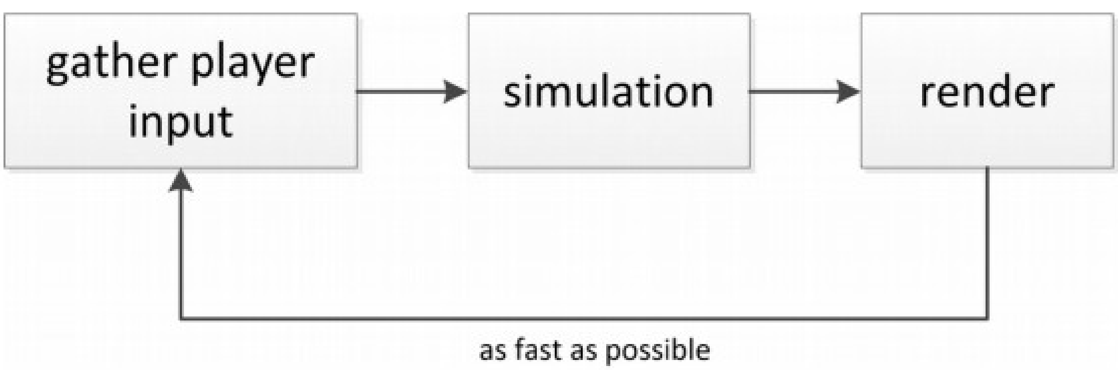

Figure 2: Simple game loop[VCF16]

2.2 Simplicity:

Just imagine: you want to drive a car home. As a driver, is it more important to you how the cylinder works inside the engine or how the oil circulates inside it? The answer is NO. The same thing is happening here, The client code which can be the Game UI or Level Manager, should not need to manage the underlying hashing logic. The interface of the game should only focus on concepts like adding or removing the tower, not low-level data manipulation.

2.3 Extensibility:

It is said that "No single platform can provide everything out of the box to meet business needs. To drive product and quality excellence, the extensibility of the chosen software platform is critical". Therefore, an extensible software platform should be flexible, configurable, customizable, upgradeable, accessiblem and collaborative[Raz24]. The game is designed to support future expansions, which dynamically add new features like towers without recompiling the core structure.

3 Core Operation:

3.1 add(item)

: With every item like tower or coin, it will add it to the inventory of the game. If that item is already available, it will count +1 to that item, if not, add will create one.

3.2 remove(item)

By using remove, from our bucket, the key and its value will be removed. In some cases, maybe the bucket become fully empty. This provides average constant-time complexity $\mathbf{O(1)}$.

3.3 count(item)

It will look up a specific item, such as arrows, gold, towers, etc. The time complexity will be $\mathbf{O(1)}$.

3.4 contains(const string& key)

It will present a true or false. If true, it means in our game inventory of the game have that item like tower or arch. If false, it is not available. So overall, It will check the availability of that item. The same as before, the time complexity will be $\mathbf{O(1)}$.

4 Set Operations:

4.1 union_with(other_items)

In this game, we have a permanent inventory. In the game context, we will have some temporary containers, such as rewards from an attack or a cleared wave. This merge_loot operation behaves like a Sum Union for multisets. Let me bring an example: if player holds Archer: 3 and obtains a loot crate with Archer:8, Cannon:1, the result will be: Archer:11, Cannon:1. It means this operation combines existing inventory with new inventory and creates a new one.

The time complexity is the number of items in the loot source. Number of items in the hash table means the number of unique types or keys. For clarification, let me bring an example: a loot chest contain 200 arrows and 3000 gold coins. It seems it will check 200 items, but the hash table sees only 2.

As a conclusion for this part, since the loot table is typically small and it is key-wise, not count-wise, combined hash tables will be extremely fast.

4.2 intersection_with(other_items)

Just imagine this is my inventory in the game: [wood:50, gold:10, fire:25, arrow: 20, diamond:7] and I want to buy a new tower as main defense system of the game. To purchase that, the player needs to pay [wood:10, silver:20, diamond:3]. In this point, intersection will tell you which items do you have and ignoring the ones player don't have or need.

But how it works as a hashtable? For each key, it perform a search in key inside the (e.g. tower: goal) hashtable. Then, for each key in goal hashtable, it will search for that key in other hashtable. if key exists, minimum of that key will be calculated. Then insert the key and min to the multiset. If that key is not available, just ignore it. Although the time complexity of hashtable is **O(1)**, here it should check and search every item in the goal hashtable, so **O(N)**.

5 Extension Feature:

In TowerDefense game I am designing, I want to make it more challenging. But how?!

The answer lies in ***remove_n(size_t n)*** which gives me option to add penalty as an arbitrary removal from the main inventory. Let me before jumping into technical details, bring an example: my inventory is as mentioned before: [wood:50, gold:10, fire:25, arrow: 20, diamond:7]. In one area of the game, some of my towers are under attack, and in one of these towers a "goblin" is stealing randomly from the inventory(ex: [wood:5, gold:2, fire: 10]) when I am not aware of that and it will continue until it finish or I attack and remove it. It makes the scenario more interesting.

How to technically implement it? I am using HashTable and this method will start to iteratre from random bucket occupied. If that item which arbitrarily chosen by "goblin" is not at the bucket, the key will be deleted. if that item is available, stolen amount will be subtracted from inventory.

6 UML Diagram / Abstraction Boundary:

I will have two interface, public interface (methods available to users) (+) and private internal members (methods used internally) (-). I design the Figure 3 with drwa.io and tried to show all the element of the TowerDefense game. Just imagine my game is like a bank, if the core of the game is public, other customers can easily come in and take the money but when it is private, I will put a teller(public part of the game) there, he will check the account and if feasible, will do the job, if not cancel the transaction.

7 Trade-off Analysis

Let's one more time check what is the **Tower Defense** game: a subgenre of strategy games where the goal is defend a player's territories or possessions by obstructing the enemy attackers or by stopping enemies from reaching the exits, usually achieved by placing defensive structures on or along their path of attack[Ree15a, Ree15b]. So based on this definition, for me performance speed has the highest priority and attacks and change of resource is happening constantly. The average time complexity of **HashTable** is **O(1)** but this item is **O(log N)** for **AVLTree**. Based on the definition and the time complexity, HashTable is a better choice.

8 Alternative Design Sketch:

HashTable is unordered, it gives me the option to, without any extra coding, use the arbitrary feature of **remove_n**$(size_t n)$ which makes the game more interesting. Now imagine I have to design the **TowerDefense** with AVLTree. AVLTree always keeps items sorted(numerically or alphabetically). So, what is the problem if using AVLTree instead of HashTable?! Again, let me bring an example for better understanding. In my inventory, I have [wood:50, gold:10, fire:25, arrow: 20, diamond:7]. AVLTree will keep them in alphabet order [arrow, diamond, fire, gold, wood]. During the game, if the enemy's goblins attack and have access to the inventory, they will first have access to an arrow. So, It will be so predictive. To add more unpredictability, I have to write extra code to generate indices

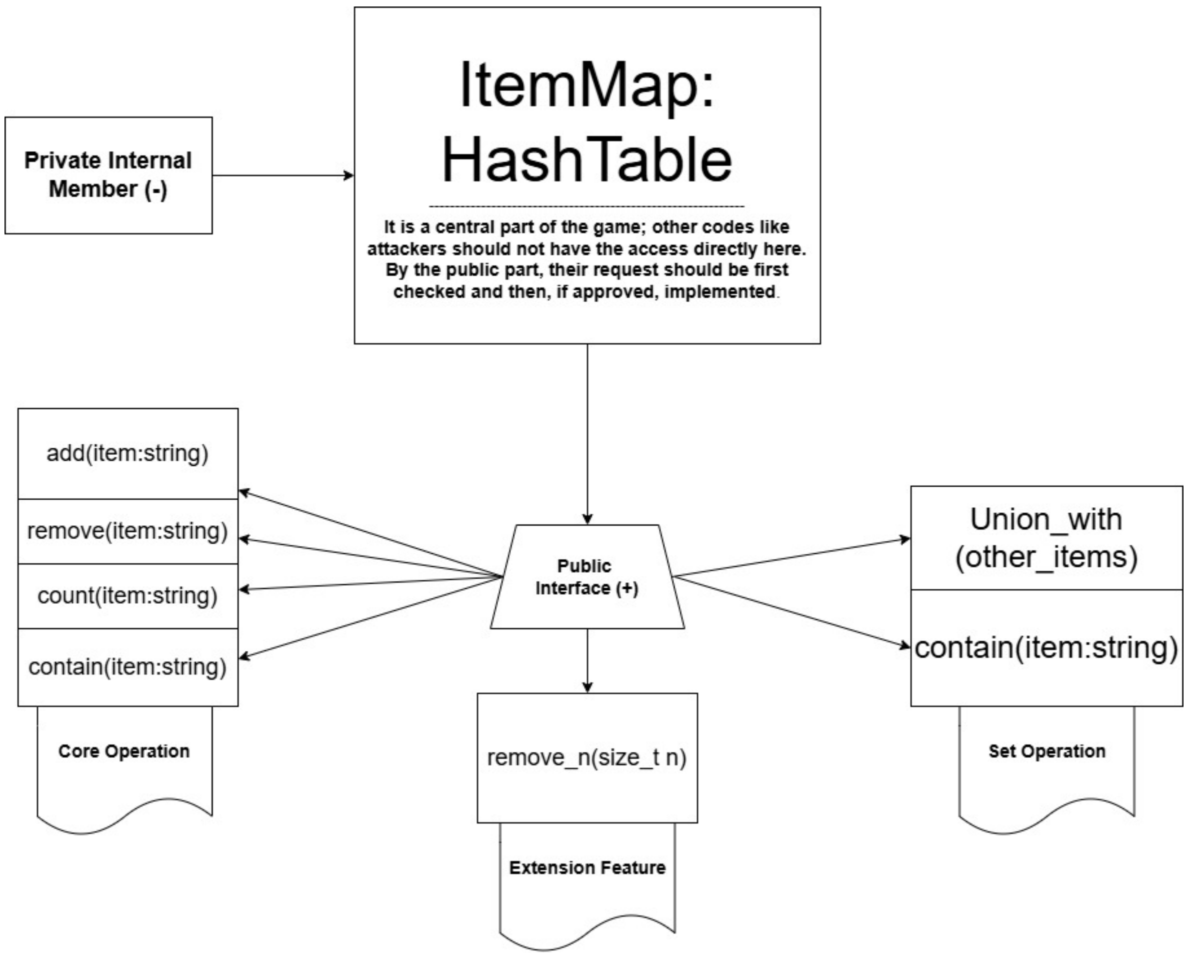

Figure 3: UML Object Diagram

randomly, which adds more complexity. Moreover, for inserting instead of **O(1)** of HashTable, in AVLTree it take **O(log N)** to find the place to insert.

In comparison of HashTable and Sequence(ex: list), when I want to add a gold, HashTable have the specific address for this item, will go there and add and update the count, DONE. In a sequence, it is long list and it should start from index zero, check every item and then update the count. Too much work, increase the latency.

9 Evaluation Plan:

For this part, I would check if **add & count** is working correctly or not? I will add 5 arrows and then again, add another 5 arrows. Then will use the count to see if the count returns 10 or not. At the next step, I will use the **remove & contain** to remove the diamond, then using contain. If it returns True, it means it is working. Also, if in my inventory is [wood:50, gold:10, fire:25, arrow: 20, diamond:7] and the player who attackes has this [gold:20, diamond: 5], implementing the **union_with()** will create this inventory: [wood:50, gold:30, fire:25, arrow: 20, diamond:12]. If it does, everything is workig as it should.

As I mentionded before, game should have a minimum latency in executing the order. I will add a large number of items, such as 1,000,000, to my inventory. Then perform 5000 add() and 30000 remove() operation. If the system works with a very short lag or latency, it shows efficiency; otherwise, it is not working as it should.

10 Conclusion:

It was my first time designing a game, although it is simple, I understood the main concepts, how to think about combining stuff together, and why one of the data structures has superiority over others. Combination of Multiset with HashTable shows the priority of speed over everything. As mentioned in Figure 3, these core, set operations and extension features are selected to show the basic concepts of designing the TowerDefense game. I tried to add some unpredictability to the game without any extra line of code to make it more exciting.

References

[Gee] GeeksforGeeks. Advantages of bst over hash table. https://www.geeksforgeeks.org/dsa/advantages-of-bst-over-hash-table/. Accessed: 2025-11-30.

[Gee24] GeeksforGeeks. Compaction in operating system. https://www.geeksforgeeks.org/operating-systems/compaction-in-operating-system/, 2024. Accessed: 2025-11-30.

[IBM] IBM. What is latency?

[Raz24] Zara Raza. What is extensibility?, March 29 2024. Accessed: 2025-11-30.

[Ree15a] Damon Reece. Best tower defense games of all time. https://www.gameranx.com/features/id/3477/article/best-tower-defense-games-of-all-time/, 2015. Accessed via Wikipedia "Tower defense" article.

[Ree15b] Damon Reece. Best tower defense games of all time. https://www.gameranx.com/features/id/3477/article/best-tower-defense-games-of-all-time/, April 2015. Archived from the original on March 29, 2016. Retrieved March 29, 2016.

[VCF16] Luis Valente, Aura Conci, and Bruno Feijo. Game loop model properties and characteristics on multi-core cpu and gpu games. In *Proceedings of SBGames*, 2016.