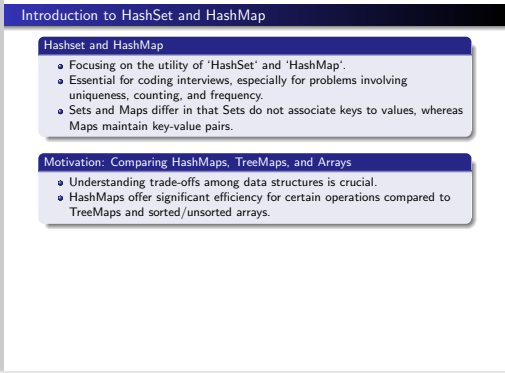

Having talked about TreeSets and TreeMaps in the previous chapter, let's discuss how the map and the set interface can be implemented using hashing. In this chapter, we will be focusing on the usage of a HashSet and a HashMap, with the next chapter covering their implementation under the hood.
Hash maps are probably one of the most useful data structures/concepts for coding interviews, for reasons we will discuss soon. They are also extremely ubiquitous in interviews. When questions ask "unique, count, frequency", take hash maps out your arsenal.

Recall that the difference between a set and a map is that in sets do not map their keys to anything, whereas maps have key-value pairs.

Time Complexity Comparison

Operation	TreeMap	HashMap	Array
Insert	$O(\log n)$	$O(1)$	$O(n)$
Remove	$O(\log n)$	$O(1)$	$O(n)$
Search	$O(\log n)$	$O(1)$	$O(\log n)$ / $O(n)$
Inorder Traversal	$O(n)$	-	-

Table: Operation Time Complexity for TreeMap, HashMap, and Array

- The listed time complexity for HashMaps is average-case.
- In interviews, it's common to assume constant time for HashMap operations.

The time complexity listed in the table for hash maps is the average case time complexity. However, it is widely accepted in interviews to assume constant time complexity.

Tree Maps vs. Hash Maps

- **Ordering:** Hash Maps do not maintain order, unlike Tree Maps or arrays.
- To iterate through Hash Map keys in a specific order, sorting is necessary, leading to $O(n \log n)$ complexity.
- **Use Case for Hash Maps:**
  - Ideal for counting frequencies or occurrences due to their key-value storage model.
  - Example: Counting names in a phonebook can be efficiently done by mapping each name to its frequency.
- While Hash Maps offer speed and efficiency for insert, remove, and search operations, the lack of inherent order is a trade-off.

Tree Maps vs Hash Maps

The downside of hash maps is that they are not ordered, so there is no guarantee that the map will store the values in set positions like BSTs or arrays do. If you were to iterate through all the keys, you would first need to sort them and then traverse, which will run in O(n log n) time.

Because hashmaps don't allow duplicates and have key-value pairs, we can use them to count frequency of keys. Going back to our phonebook example, we can count the number of times a given name appears in our phonebook by mapping the name to its frequency.

Counting Frequencies with HashMap

- Utilizing a **HashMap** efficiently counts the occurrences of elements in an array.
- **Example Array:** ["*alice*","*brad*","*collin*","*brad*","*dylan*","*kim*"]
- In a HashMap:

Key	Value
Alice	1
Brad	2
Collin	1
Dylan	1
Kim	1

- **Algorithm:**
  - If a name is encountered for the first time, add it to the HashMap with a frequency of 1.
  - If a name already exists, increment its frequency by 1.

Efficiency

- TreeMap: Insertion operation costs $O(\log n)$, leading to $O(n \log n)$ for all insertions.
- HashMap: Overall operation costs $O(n)$, showcasing its efficiency over TreeMap for this task.

```
names = ["alice", "brad", "collin", "brad", "dylan", "kim"]

countMap = {}
for name in names:
    # If countMap does not contain name
    if name not in countMap:
        countMap[name] = 1
    else:
        countMap[name] += 1
```