Use Cases

1 When to use which Data Structure?

As you know from reading this section, there are quite a few data structures. When tackling design questions, it can be overwhelming to think about which data structure you need to use. We will briefly discuss the use cases for the various categories of data structures.

Lists are generally used when you need to just store data in some ordered manner. Runtime isn't too important when you're using a list. The two main types of lists are **LinkedLists** and **ArrayLists**.

1. **LinkedList**
   - get() = $O(N)$
   - add() = $\Theta(1)$
   - remove() = $\Theta(N)$
   - space = $O(N)$
2. **ArrayList**
   - get() = $O(1)$
   - add() = $\Theta(N)$
   - remove() = $\Theta(N)$
   - space = $O(N)$

Sets are generally used when you have unordered data. In sets, you are not allowed to have any duplicates and you can only store keys! Essentially, sets are just maps with some dummy value. Data Structures that use sets are used when you just need to store elements.

Maps are data structures that store a key and a value. If you insert a key value pair into a map where that key already exists, you replace that key's value with the value of the key value pair you are inserting. You declare Maps in the following manner:

```
Map<Key type, Value type>
```

In maps, you have an immutable key and values that can be any data type. Frequently, maps can be used to see how frequently a key turns up.

Trees can be used whenever the keys in questions are implementing comparable. Trees can be used when you are attempting to find some sort of order. Specific types of **self balancing** trees are **2-3 Trees** and **Left Leaning Red Black Trees**. You would choose to use trees over hashing when the key is hard to hash. This can occur when a key is very long, and would take a lot of time to perform arithmetic on it or analyze it.

1. **Binary Search Tree**
   - search() = Average: $O(logN)$ Worst Case: $O(N)$
   - insert() = Average: $O(logN)$ Worst Case: $O(N)$
   - delete() = Average: $O(logN)$ Worst Case: $O(N)$
   - space = $O(N)$
2. **Balanced Search Trees (2-3 Trees and Left Leaning Red Black Trees)**
   - get() = $\Theta(logN)$

- insert() = $\Theta(logN)$
- remove() = $\Theta(logN)$
- space = $O(N)$

Hashing data structures are used when you need $\Theta(1)$ runtime for all operations assuming that you have a good hashcode. You would also want to use hashing data sets when your data is unordered.

1. **Balanced Search Trees (2-3 Trees and Left Leaning Red Black Trees)**
   - get() = Average: $O(1)$ Worst Case: $O(N)$
   - insert() = Average: $O(1)$ Worst Case: $O(N)$
   - remove() =Average: $O(1)$ Worst Case: $O(N)$
   - space = $O(N)$

Stacks are used when you want to look at the most recent thing that has been done. Stacks are known as first in last out data types (FILO). This can be applied to search history, delivery items etc. Stacks are often used in depth first search in trees and also graphs, which we will go over in the next chapter.

1. **Stack**
   - push() = $\Theta(1)$
   - pop() = $\Theta(1)$
   - peek() = $\Theta(1)$
   - space() = $O(N)$

Queues are used when we need to view items by the order in which they were done. Queues are known as first in last out data types (FIFO). In the real world, this is used when we have to keep track of waitlists. Additionally Queues are used for breadth first search in trees as well as graphs.

1. **Queue**
   - add() = $\Theta(1)$
   - remove() = $\Theta(1)$
   - peek() = $\Theta(1)$

Heaps or Priority Queues can be used whenever the keys we are examining implement Comparable. Anytime you need the most or least of something, use a Heap. These are not for overall order, unless it's taking out one item at a time.

1. **Heap**
   - search() = $\Theta(1)$
   - insert() = Average: $O(1)$ Worst Case: $O(logN)$
   - delete() = $O(log(N))$
   - peek() = $O(1)$
   - space = $O(N)$

Tries are used when you want to perform searches and insertions using prefixes. There are 2 types of tries, normal tries and **ternary search tries**. Normal tries have a very fast speed but in exchange they take up a lot of space. Ternary search tries on the other hand are relatively slow but take up much less space.

1. **Tries**
   - M is the length of the string you are attempting to insert/search for, N is amount of keys, L is the length of the longest key, and R is the size of the alphabet.

- search() = $\Theta(M)$
- insert() = $\Theta(M)$
- space = $O(N * L * R)$

2. **Ternary Search Tries**

- N is the amount of keys and L is the length of the longest key
- search() = $\Theta(N)$
- insert() = $\Theta(N)$
- space = $O(N * L)$