Abstract Data Structures

ArrayLists vs HashSets

Task 1

On the unit page, let's take a moment to play with the simulator and observe what is happening when we enter a String (or any object) into an ArrayList or a Hash Table, both Abstract Data Types.

Insert the following three words into the simulator, one by one:

As we would expect, in an ArrayList, the first word will be stored at index 0 of the **list**, the second word at index 1 and so on.

But in a program which stores data in a hash table , *the data itself* is used to determine which *bucket* the data will be inserted into the **hash table**. To determine the exact bucket, a **hash function** is used.

Task 2

Look at the log. Write a paragraph describing the **hash function** used in this simulator to determine the bucket at which the data will be stored in the hash table:

Task 3

Click the **reset** button and enter the following strings into the simulator:

Analyze the log:

```
--------------------------------
INSERTING: hello
ASCII Sum = 532
532 % 10 = 2
Stored in Bucket 2
--------------------------------
INSERTING: umbrella
ASCII Sum = 852
852 % 10 = 2
Collision at Bucket 2
Linear Probe → Bucket 3
Stored in Bucket 3
```

The hashing function produced 2 for “hello” and inserted the String into Bucket #2 of the Hash Table. But “umbrella” produced the same bucket number! This is known as a **collision**. Somehow, the table had to resolve this collision issue and it used a strategy called **linear probing**. Do some research and explain this collision resolution strategy here:

Task 4

Click the **reset** button and enter the following strings into the simulator:

As before, the ArrayList orders the data by index. The program performs a hashing function on the data itself to determine the location/bucket/index in the hash table. We can see that a hash table is, by nature, an **unordered** data structure.

Now **search** both the ArrayList and the hash table for the word “sat” and analyse the log. How many iterations on the ArrayList were performed to find the word, how many iterations on the hash table were required, and **what do you think** the Big O time complexity is for a search on both data structures? Complete the table:

Number of iterations		**Big O Time Complexity**
ArrayList		
Hash Table		

Do some research on Big O Time complexity for searching a Hash Table. Is the thinking you entered in the table above the same as your research? Enter any notes here:

Task 5

A hash table is a data structure used by the Abstract Data Type **HashSet** which we will study in a subsequent unit. In modern languages like Java, when we do something like this:

```java
Set<String> myHashSet = new HashSet<>();
```

Java will initialise a Hash Table of (something like) 16 buckets.

When the table gets too full, *the opportunity for collisions* to occur increases and the performance of the table is reduced. Java will have to **resize** the table..

A **load factor** is used to help determine when to resize the table:

Load factor = capacity / number of entries

Let's say a hash table has a **capacity** of 16 buckets and there are currently 8 **entries** in the hash table:

Load factor = 8/16 = 0.5

When the load factor reaches a threshold, the table will be resized. The threshold is generally in the region of 0.75

So, if ***Load Factor > Load Factor threshold***, resize the table.

So when the 13th bucket becomes occupied, the Hash Table is resized.

This requires:

- Creating a larger table eg with 32 buckets (double the current capacity)
- Recalculating the bucket location for every existing entry
- Moving each existing item to its new bucket location in the new table

This is known as **rehashing**.

REVIEW

A Hash Table can be used when **fast, direct access** is required to insert, remove and search for entries. The potential for collisions must be kept low because collisions impact the performance of the table, which should ideally be O(1) for inserting, removing and searching.

The hashing function used in this scenario was:

- SUM the ASCII code of each character in the String
- Bucket Number (index) = SUM%10

This is an example of a **poor hashing function** because it will result in too many collisions. Better hashing functions exist!

Linear Probing is a Collision Resolution Strategy and in a worst case scenario can result in a time complexity of **O(n)** when searching or inserting an item into the table. The ideal hash table will have a time complexity of **O(1)**.