**ST10375530**

**Ayana Modise**

**PROGRAMMING 3B**

**[PROG7312]**

**ICE TASK 2**

**PART 1 – THEORECTICAL QUESTIONS (WRITTEN)**

1. **Differentiate Dictionary from Set in C#**

A Dictionary and a Set are both collection types in C#, but they serve different purposes and have distinct characteristics. A Dictionary is a collection of key-value pairs where each unique key maps to a specific value, allowing you to retrieve values based on their associated keys (Albahari and Albahari, 2022). In contrast, a Set, implemented through the HashSet class, stores only unique values without any associated keys or mapping (Troelsen and Japikse, 2021).

The primary difference lies in their data structure: dictionaries maintain relationships between keys and values, making them ideal for lookups and associations, whilst sets focus solely on maintaining a collection of distinct elements. For instance, if you need to store student IDs alongside their names, a Dictionary would be appropriate, but if you only need to track which course codes are being offered without additional information, a Set would suffice.

1. **Why is uniqueness of dictionary keys necessary? What happens with duplicate keys?**

The uniqueness of dictionary keys is essential because the Dictionary class uses keys as identifiers to locate and retrieve their corresponding values efficiently (Price, 2020). Each key acts as a unique address in the dictionary's internal hash table structure. If duplicate keys were allowed, the system would face ambiguity when attempting to retrieve a value, as it would not know which of the duplicate keys' values to return.

When you attempt to add a duplicate key to a Dictionary using the Add method, the system throws an ArgumentException with a message indicating that an item with the same key has already been added (Microsoft, 2024). However, if you use the indexer syntax (dictionary[key] = value), the existing value associated with that key will simply be overwritten with the new value, rather than throwing an exception. This behaviour ensures data integrity whilst providing flexibility in how you update dictionary entries.

1. **Realistic scenarios where a Set is preferable to a Dictionary**

Sets are particularly useful in several practical scenarios. Firstly, when tracking unique email addresses for a mailing list, you only need to ensure each email appears once without requiring additional associated data (Stellman and Greene, 2020). A HashSet prevents duplicate subscriptions efficiently without the overhead of maintaining key-value pairs.

Secondly, in educational settings, tracking unique course codes that a student has enrolled in requires only the course identifiers themselves. There is no need for associated values, making a Set the natural choice. Thirdly, when implementing tag systems for blog posts or articles, you want to ensure each tag appears only once per item. A HashSet provides this functionality with optimal performance for checking whether a tag already exists before adding it.

1. **How does HashSet guarantee uniqueness?**

The HashSet class guarantees uniqueness through its implementation of hash-based storage combined with equality comparison (Albahari and Albahari, 2022). When you add an element to a HashSet, it first computes a hash code for that element using the GetHashCode method. This hash code determines which internal bucket should store the element.

Before actually adding the element, the HashSet checks whether an equal element already exists in that bucket by using the Equals method (Troelsen and Japikse, 2021). If an element with the same hash code and equality comparison is found, the new element is rejected and the Add method returns false, indicating the element was not added. This two-step process of hashing and equality checking ensures that duplicate elements cannot exist within the collection, maintaining the mathematical properties of a set.

1. **Performance differences between Dictionary and List lookups**

Dictionary lookups offer significantly better performance compared to List lookups due to fundamental differences in their search mechanisms (Price, 2020). A Dictionary uses hash-based lookup, providing O (1) average time complexity, meaning the lookup time remains constant regardless of the number of elements stored. The dictionary computes the hash code of the search key and directly accesses the corresponding bucket location.

In contrast, a List requires sequential searching through elements, resulting in O(n) time complexity, where n represents the number of elements (Stellman and Greene, 2020). As the list grows larger, lookup times increase proportionally. For example, finding a specific student ID in a Dictionary of 10,000 students takes roughly the same time as finding one in a Dictionary of 100 students. However, searching through a List of 10,000 students could take up to 100 times longer than searching a list of 100 students. This makes dictionaries far superior for scenarios requiring frequent lookups based on specific identifiers.

**Part 2: Practical Coding Tasks (VS Console App)**

GitHub Repository Link: <https://github.com/ST10375530/PROG7312_ICETASK2.git>

**Reference List**

* Albahari, J. and Albahari, B. 2022. C# 10 in a nutshell: The definitive reference. Sebastopol: O'Reilly Media.
* Microsoft. 2024. Dictionary<TKey,TValue> class. [Online]. Available at: <https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2> [Accessed 6 October 2025].
* Price, M.J. 2020. C# 9 and .NET 5: Modern cross-platform development. 5th ed. Birmingham: Packt Publishing.
* Stellman, A. and Greene, J. 2020. Head first C#: A learner's guide to real-world programming with C# and .NET Core. 4th ed. Sebastopol: O'Reilly Media.
* Troelsen, A. and Japikse, P. 2021. Pro C# 9 with .NET 5: Foundational principles and practices in programming. 10th ed. New York: Apress.