2	Hashtables, Queue und Stack

Was versteht man unter Hashtables

Eine Hashtable ist eine Datenstruktur, die dazu dient, Daten in Form von Schlüssel-Wert-Paaren effizient zu speichern und abzurufen. Schlüssel werden durch eine Hashfunktion in Indizes umgewandelt, die auf Speicherplätze in einem Array zeigen. Bei Kollisionen, wenn mehrere Schlüssel denselben Index erhalten, werden die entsprechenden Werte in einer Struktur wie einer Liste gespeichert.

Unterschied zwischen Hashtable und Array

Hashtables und Arrays ähneln sich darin, dass beide Datenstrukturen für das Speichern und Abrufen von Elementen genutzt werden und schnellen Zugriff auf gespeicherte Daten ermöglichen. Beide verwenden interne Speicherplätze, um Elemente effizient zu organisieren.
Der Hauptunterschied liegt in der Zugriffsweise: Bei Arrays erfolgt der Zugriff über numerische Indizes, während Hashtables Schlüssel verwenden, die durch eine Hashfunktion einem Speicherplatz zugeordnet werden. Außerdem haben Arrays eine feste Größe und eine geordnete Struktur, während Hashtables dynamisch wachsen können und keine feste Ordnung haben.

```
// Eine Hashtable erstellen
Hashtable hashtable = new
Hashtable();

// Schlüssel & Wert hinzufügen
hashtable.Add("Name", "Max
Mustermann");

// Zugriff auf Werte
Console.WriteLine("Name: " +
hashtable["Name"]);

// Ein Schlüssel & Wert
entfernen
hashtable.Remove("Beruf");

//Iteration durch Einträge
foreach (DictionaryEntry entry
in hashtable) {
Console.WriteLine($"Schlüssel:
{entry.Key}, Wert:
{entry.Value}"); }
```

Queue

Eine Queue ist eine Datenstruktur, die das **FIFO-Prinzip** (First In, First Out) verwendet. Elemente werden am Ende der Warteschlange hinzugefügt und am Anfang entfernt. Dies ermöglicht eine geregelte Verarbeitung, bei der das zuerst hinzugefügte Element auch als erstes wieder entfernt wird. Typische Operationen einer Queue sind Enqueue() zum Hinzufügen von Elementen und Dequeue() zum Entfernen und Zurückgeben des ersten Elements. Weitere Varianten von Warteschlangen, wie z.B. threadsichere, bieten zusätzliche Mechanismen für den Zugriff in parallelen Umgebungen.

```
// Eine Queue erstellen
Queue<string> queue = new Queue<string>();

// Elemente in die Queue einfügen
(Enqueue) queue.Enqueue("Erstes Element");

// Zugriff auf das erste Element ohne es zu entfernen (Peek)
Console.WriteLine($"\nPeek (Erstes Element): {queue.Peek()}");

// Entfernen von Elementen aus der Queue (Dequeue)
Console.WriteLine($"\nElement entfernt: {queue.Dequeue()}");

 // Anzahl der verbleibenden Elemente
Console.WriteLine($"\nAnzahl der Elemente in der Queue:
{queue.Count}");
```

Stack

Die **Stack**-Datenstruktur folgt dem **LIFO-Prinzip** (Last In, First Out), bei dem das zuletzt eingefügte Element als erstes entfernt wird. Mit der Methode Push() können Elemente hinzugefügt werden, während Pop() das oberste Element entfernt und zurückgibt. Die Methode Peek() gibt das oberste Element zurück, ohne es zu entfernen. Stacks werden häufig für rekursive Prozesse oder die Rückverfolgung von Aufgaben verwendet. Für eine threadsichere Nutzung kann eine synchronisierte Version erstellt werden.

```
// Einen Stack erstellen
Stack<string> stack = new Stack<string>();

// Elemente auf den Stack legen (Push)
stack.Push("Erstes Element");

// Zugriff auf das oberste Element ohne es zu entfernen (Peek)
Console.WriteLine($"\nPeek (Oberstes Element):
{stack.Peek()}");

// Elemente vom Stack entfernen (Pop)
Console.WriteLine($"\nElement entfernt: {stack.Pop()}");

// Anzahl der verbleibenden Elemente
Console.WriteLine($"\nAnzahl der Elemente im Stack:
{stack.Count}");
```

Quellen: https://learn.microsoft.com/de-de/dotnet/api/system.collections.hashtable?view=net-8.0
https://learn.microsoft.com/de-de/dotnet/api/system.collections.queue?view=net-8.0
https://learn.microsoft.com/de-de/dotnet/api/system.collections.stack?view=net-8.0