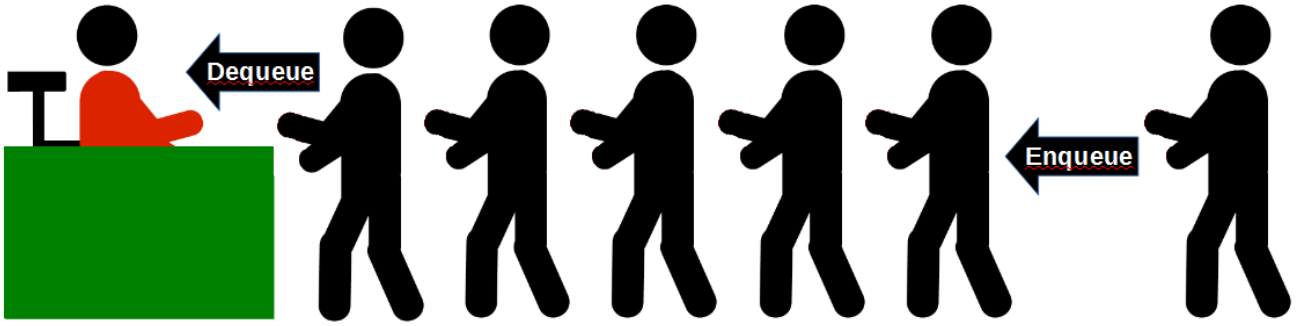
Queue (Schlange)

Ein weiterer grundlegender Container und ADT ist die Queue oder (Warte-)Schlange. Vorstellen kann man sich diese wie eine Schlange an einer Kasse im Supermarkt.



Neue Elemente werden prinzipiell am Ende der Warteschlange angehängt und vorn am Beginn der Warteschlange abgearbeitet.

Man nennt dieses Prinzip daher auch **First-In-First-Out** (FIFO).

Umgangssprachlich kann man auch sagen: Wer zuerst kommt, mahlt zuerst.[1]

Die Implementation einer Queue kann auf verschiedene Weisen erfolgen. Zum Beispiel kann das Prinzip einer verketteten Liste verwendet werden. Man kann aber auch ein Array nutzen. Das kann dann sowohl ein dynamisches (d.h. in der Größe veränderliches) Array sein. Genau so gut, kann aber auch ein statisches Array (mit einer festen Größe) verwendet werden. Sollte hierbei allerdings dessen Limit erreicht werden, muss ggf. ein neues größeres Array erzeugt werden und es müssen alle bisherigen Array-Elemente in das neue Array kopiert werden. Hier entscheidet letztendlich der jeweilige Einsatzzweck (z.B. ändert sich die Anzahl der Elemente sehr häufig) auf welcher Basis die Queue programmiert wird.

Eine Queue kennt typischerweise nur zwei Operationen zum Bearbeiten des Datenbestandes:

- **`enqueue:`** Das Einfügen eines neuen Elements am Ende der Queue.
- **`dequeue:`** Das Entnehmen eines Elements (zur Verarbeitung) am Beginn der Queue.

Die in der Schlange hinter dem ersten Element noch „wartenden“ Elemente sind normalerweise nicht erreichbar.

Eine weitere Methode, die in einer Queue-Implementation zu finden ist, heißt zumeist **`peek`**. Diese liefert das erste Element, ohne es aber zu entnehmen. Ebenfalls denkbar wäre eine Methode **`ClearQueue`**, die alle Elemente in der Queue löscht. Zuletzt noch eine Methode, die direkt Auskunft über den Status einer Queue Auskunft gibt: **`IsEmpty():bool`**. Ist die Schlange leer, liefert diese Methode den Wert **`true`**, ansonsten **`false`**.

Anwendungen

Queues finden in verschiedenen Kontexten ihre Anwendung. Oftmals werden Queues verwendet um eine asynchrone Kommunikation zu ermöglichen. Wenn z.B. von mehreren Prozessen Daten zu versenden sind, müssten diese Prozesse bei synchroner Kommunikation mit dem Empfänger u.U. auf diesen warten, bis er eben frei für den jeweiligen Sender ist.

Durch eine Queue können diese Blockierungen gelöst werden: Die Sender stellen ihre Nachrichten in eine Sender-Queue, die kontinuierlich abgearbeitet wird, solange Elemente enthalten sind.

1 Das „mahlen“ kommt von „Mahlen von Mehl beim Müller“ und hat nichts mit dem „Malen eines Bildes“ zu tun!

Beim Empfänger, der ebenfalls u.U. mehrere Prozesse zu bedienen hat, werden die übertragenen Daten ebenfalls erst in eine Empfänger-Queue gestellt und dann kontinuierlich an die jeweiligen Empfangsprozesse übermittelt. Sowohl Sender als auch Empfänger sind in der Verarbeitung der Daten zeitlich entkoppelt.

In der Informatik ist dies bekannt als Erzeuger-Verbraucher-Problem (Producer-Consumer-Problem).

Auch unterschiedliche Verarbeitungsgeschwindigkeiten können so ausgeglichen werden. Beispielsweise schreibt ein Programm nicht direkt auf einen Datenträger (z.B. eine Festplatte), sondern in einen Pufferspeicher, der dann sukzessive auf den Datenträger übertragen wird. Dies passiert auch in umgekehrter Richtung: Der Datenträger schreibt die Daten in einen Pufferspeicher, aus dem dann die verschiedenen lesenden Prozesse die Daten entnehmen können

Ein Printserver empfängt zu druckende Daten von mehreren Benutzern eines Druckers. Jeder empfangene Druckjob wird in eine Printerqueue gestellt und dann einer nach dem anderen zum Drucker gesendet.

Queues finden ebenfalls bei der sogenannten Interprozesskommunikation Anwendung. Zwei Prozesse auf einem Rechnersystem kommunizieren miteinander per Queue. Der sendende Prozess stellt seine Daten in eine Queue, aus der der empfangende Prozess sie dann entnimmt.

Weiterhin gibt es neben der „klassischen“ Queue auch sogenannte Priority-Queues. Bei diesen Prioritätswarteschlangen hat jeder Eintrag eine gewisse Priorität, die bei der Abarbeitung berücksichtigt wird. Wichtigere Jobs werden gegenüber den unwichtigen Jobs vorgezogen.

Implementation einer Queue<T>

In diesem Beispiel soll die generische Klasse **`Queue<T>`** ebenfalls mit Hilfe einer verketteten Liste implementiert werden.

Sie besteht aus der äußeren Klasse SimpleQueue, die die Operationen der Queue implementiert und die Verwaltungsinformationen **`head`**, **`tail`** und **`Count`** enthält.

Weiterhin ist die Klasse **`SimpleQueueNode`** in diesem Beispiel in der Klasse **`SimpleQueue`** enthalten und existiert nicht als eigene Datei.

Übungen

1. Analysieren Sie die Beispiel-Implementation auf der nachfolgenden Seite.

2. Erstellen Sie ein Testprogramm, um die momentan implementierten Methoden zu testen.

3. Ergänzen Sie die in der Klasse **`SimpleQueue`** die Methode **`Dequeue():T`** zum Entfernen des ersten Elements. Testen Sie die Methode.

4. Ergänzen Sie ein Property **`IsEmpty`**, welches den Zustand der Schlange anzeigt.

5. Ergänzen Sie eine Methode **`ClearQueue():void`**, die alle Elemente entfernt. Nutzen Sie hierzu die Methode **`Dequeue()`**. Testen Sie die Methode.

6. Informieren Sie sich über das Prinzip eines Ringpuffers als Variante einer Queue.

```
public class SimpleQueue<T>
{
    // ************************************
    // Eine Klasse, die nur innerhalb von SimpleQueue verwendet wird
    // Ein Queue-Element
    public class SimpleQueueNode<X>
    {
        public T Data { get; set; }
        public SimpleQueueNode<X> Next { get; set; }

        public SimpleQueueNode()
        {
            Data = default(T);
            Next = null;
        }
        public SimpleQueueNode(T data)
        {
            Data = data;
            Next = null;
        }
    }
    // Ende der Klassendefinition
    // ************************************

    // Attribute von SimpleQueue
    public int Count;
    private SimpleQueueNode<T> head { get; set; }
    private SimpleQueueNode<T> tail { get; set; }

    // Konstruktor für SimpleQueue
    public SimpleQueue()
    {
        Count = 0;
        head = new SimpleQueueNode<T>();
        tail = new SimpleQueueNode<T>();
    }

    public void Enqueue(T data)
    {
        // Neues Queue-Element erzeugen
        SimpleQueueNode<T> tmp = new SimpleQueueNode<T>(data);

        // Zwei Fälle:
        // 1. Queue ist leer
        // 2. Queue ist nicht leer

        if (Count == 0)
            head.Next = tail.Next = tmp;
        else
        {
            tail.Next.Next = tmp;
            tail.Next = tmp;
        }
        Count++;
    }

    public T Peek()
    {
        SimpleQueueNode<T> rc = new SimpleQueueNode<T>();

        // Zwei Fälle:
        // 1. Queue ist leer
        // 2. Queue ist nicht leer
        if (Count > 0)
           rc = head.Next;
        return rc.Data;
    }
}
```