LERNTAGEBUCH

Data-Structures & Algorithms

Inhaltsverzeichnis

[**DATA STRUCTURES** 2](#_Toc70671001)

[ARRAYS 2](#_Toc70671002)

[Code 4](#_Toc70671003)

[Eindimensionale Arrays 4](#_Toc70671004)

[Multidimensinale arrays 4](#_Toc70671005)

[Jagged arrays (verzweigte Arrays) 5](#_Toc70671006)

[Linked Lists 7](#_Toc70671007)

[Bestandteile 7](#_Toc70671008)

[Suchen 7](#_Toc70671009)

[Vorteile 7](#_Toc70671010)

[Doppelt Verkettete Liste 8](#_Toc70671011)

[Kreisförmig verkettete Liste 9](#_Toc70671012)

[Code 9](#_Toc70671013)

[Stacks 11](#_Toc70671014)

[Push 11](#_Toc70671015)

[Pop 11](#_Toc70671016)

[Code 11](#_Toc70671017)

[Queues 13](#_Toc70671018)

[Enqueue 13](#_Toc70671019)

[Dequeue 13](#_Toc70671020)

[Code 13](#_Toc70671021)

[Hash Tables 15](#_Toc70671022)

[Trees 15](#_Toc70671023)

[Heaps 15](#_Toc70671024)

[Graphs 15](#_Toc70671025)

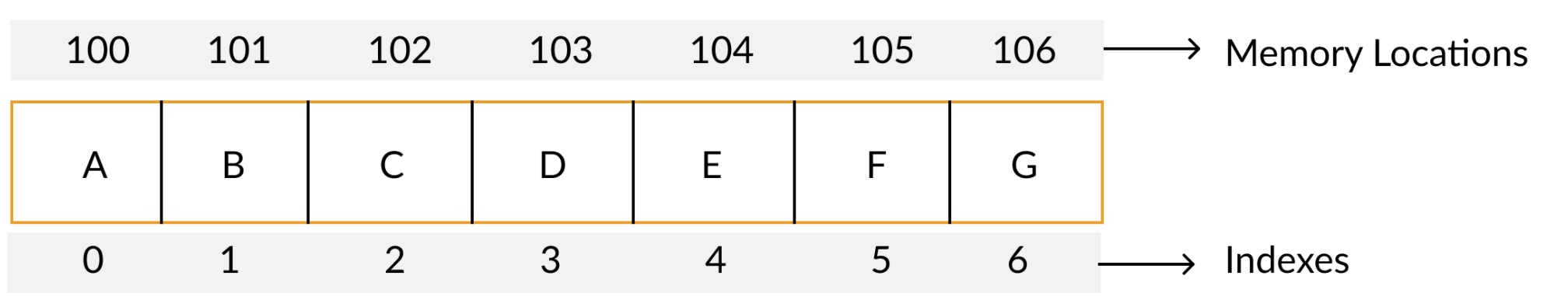
[**ALGORITHMS** 15](#_Toc70671026)

[**REFERENZEN** 16](#_Toc70671027)

# **DATA STRUCTURES**

Source-Code: <https://github.com/iKirbs/DataStructures>

## ARRAYS[[1]](#footnote-1) [[2]](#footnote-2)



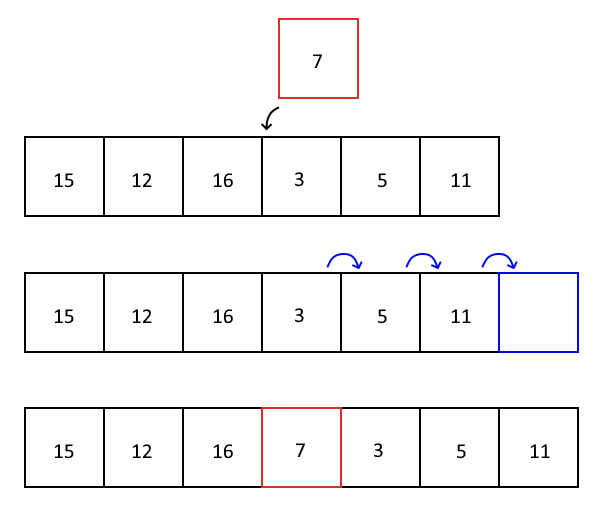
Arrays haben eine vorgegebene Größe und reservieren bei Deklaration einen kontinuierlichen Block im RAM.

**Felder Einfügen**

Neue Felder innerhalb des Arrays zu erstellen ist sehr aufwändig, da ein neues Array erstellt wird, um die alten Einträge in das neue Array mit anderer Größe zu kopieren.

Alternativ gibt es hierzu **dynamic Arrays**. Dabei wird hinten am Array neuer Platz für die neuen Einträge erstellt und alle Einträge ab der Position, wo man die Felder einfügen will, werden nach hinten verschoben. Der Aufwand dieses Vorgangs hängt davon ab, wie weit vorne Felder eingefügt werden.

Der Vorgang benötigt **O(n)** Zeit.



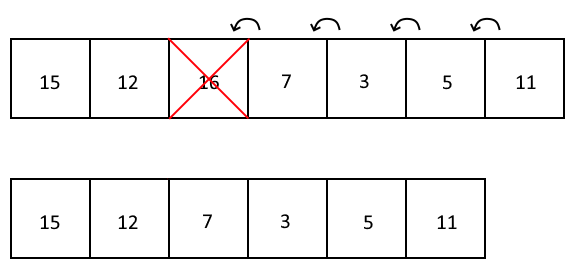
Normalerweise wird, um nicht bei jedem Insert neue Felder zu generieren, im vornherein mehr Platz reserviert als eigentlich gebraucht wird.

Man sollte beachten, dass der RAM hinter dem Array nicht gebraucht wird und somit keinen Daten überschreibt. Falls Daten vorhanden sind, muss das Array wieder mit hohem Aufwand kopiert werden.

**Felder Löschen**

Felder aus einem Array zu Löschen kann ebenfalls gemacht werden, indem man das Array ohne den zu löschenden Feldern kopiert.

Alternativ werden beim **dynamic Array** die Einträge, die rechts von der zu löschenden Stelle nach links verschoben.



**Felder Auslesen**

Das Auslesen eines Index erfolgt direkt, da die gewünschte Position im RAM berechnet werden kann. Der Zeitaufwand ist konstant, deswegen:

Lookup Time = **O(1)**

### Code

### Eindimensionale Arrays

**Arraydeklarierung**

Neues int Array erstellen mit 3 Feldern

int[] array1 = new int[3];

Felder befüllen.

array1[0] = 1;

array1[1] = 2;

array1[2] = 3;

**Arrayinitialisierung**

Bei der Arraydeklarierung kann das Array gleichzeitig initialisiert werden.

int[] array2 = new int[] { 1, 2, 3 };

**Implizit typisierte Arrays**

int[] array3 = { 1, 2, 3 };

### Multidimensinale arrays

Multidimensionale Arrays haben im Beispiel eines 2-dimensionalem Array eine vorgegebene Anzahl von Spalten und Zeilen.

**Arraydeklarierung**

int[,] array2D\_1 = new int[2, 2];

**Arrayinitialisierung**

array2D\_1[0, 0] = 1;

array2D\_1[0, 1] = 2;

array2D\_1[1, 0] = 3;

array2D\_1[1, 1] = 4;

int[,] array2D\_2 = new int[2, 2] { { 1, 2 }, { 3, 4 } };

int[,] array2D\_3 = { { 1, 2 }, { 3, 4 } };

### Jagged arrays (verzweigte Arrays)

Ein Jagged Array ist ein Array, dessen Elemente weitere Arrays sind. Die Sub-Arrays können beliebiger Größe sein.

**Arraydeklarierung**

int[][] jagged\_array = new int[2][];

jagged\_array[0] = new int[] { 1, 2 };

jagged\_array[1] = new int[] { 1, 2, 3 };

**Code Beispiel**

int[][] jagged\_array = new int[2][];

jagged\_array[0] = new int[] { 1, 2 };

jagged\_array[1] = new int[] { 1, 2, 3, 4, 5 };

Console.WriteLine("Jagged Arrays:");

foreach (var array in jagged\_array)

{

Console.WriteLine(String.Join(", ", array));

}

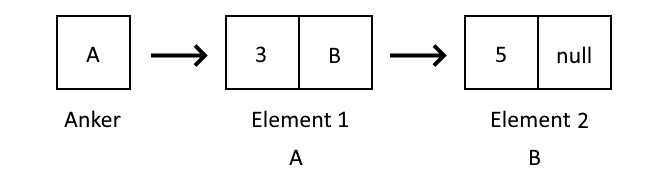
**Output**

Jagged Arrays:

1, 2

1, 2, 3, 4, 5

## Linked Lists[[3]](#footnote-3) [[4]](#footnote-4)



Die Linked List besteht aus verketteten Elementen. Dabei unterscheidet man im Allgemeinen drei Arten: **Anker, Konten, Ende.**

### Bestandteile

Jedes der Elemente hat dabei den eigenen **Wert** sowie einen **Zeiger**, der auf das **nächste** **Element** in der Liste zeigt (einfach verkettete Liste).

### Suchen

Will man nun auf das **3. Element** der Liste zugreifen, so muss man zuerst auslesen, wo die Liste beginnt (**Anker**). Der Anker zeigt dann auf den Kopf (**Element 1**), der Kopf verweist auf das nächste Element (**Element 2**) und Element 2 verweist dann auf das **3. Element.**

Anders zum Array lassen sich die RAM Positionen **nicht direkt** berechnen, sondern die Liste wird von **vorne** nach **hinten** durchsucht, bis der gewünschte Eintrag gefunden wurde.

Das Ende der Liste ist dann erreicht, wenn der Zeiger vom momentanen Element **null** ist.

### Vorteile

**Größe**

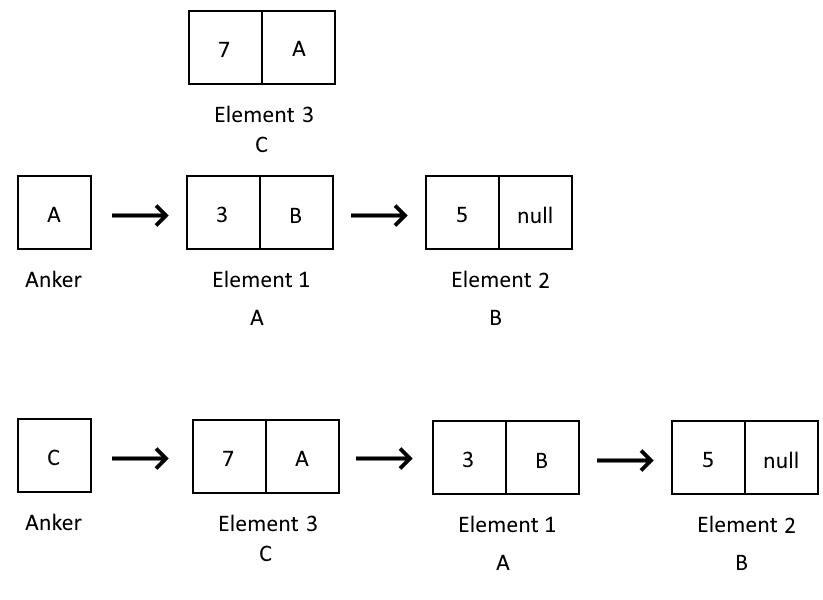
Ein Vorteil der Linked List ist, dass die Listengröße **nicht** bekannt gegeben werden muss. Falls man ein weiteres Element der Liste anfügen will, so kann man zur **Runtime** im RAM an einer freien Stelle einen neuen Platz reservieren.

**Einfügen**

Um das neue Elemente der Liste anzuhängen, muss der **Zeiger** des **letzten** **Elements** der Liste auf den **neuen** **Eintrag** verweisen. Nun ist der neue Eintrag das letzte Element der Liste. Aus diesem Grund wird die Liste auch als **dynamische** **Datenstruktur** bezeichnet.

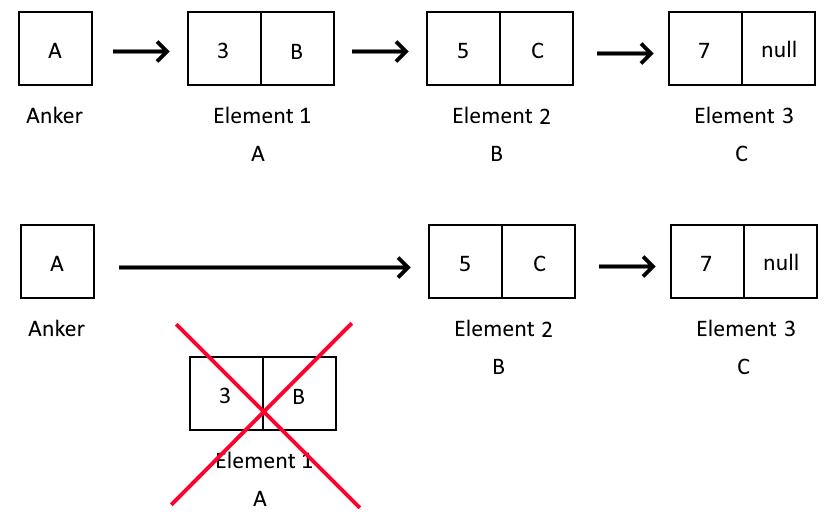
Will man an innerhalb der Liste an Stelle x ein neues Element einfügen, so muss der Zeiger, der auf Element x zeigt auf das neue Element zeigen und das neue Element muss auf Element x zeigen.

**Beispiel**: Element an Position 1 einfügen (Index 0)



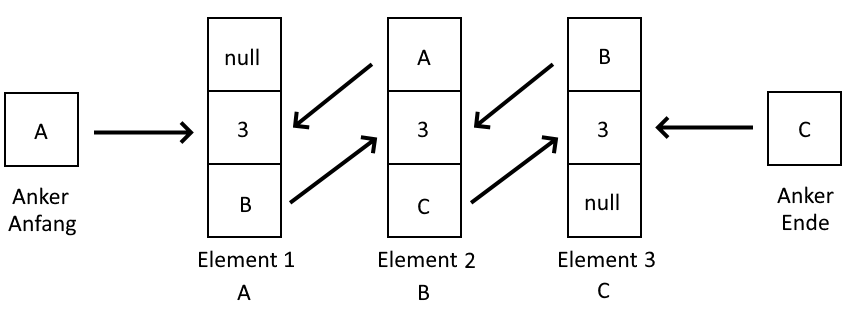
**Löschen**

Um einen Eintrag an Stelle x zu löschen, muss der Zeiger, der auf Element x zeigt, nun auf das nächste Element Zeigen. Das alte Element x kann dann gelöscht werden.



### Doppelt Verkettete Liste

Allerdings gibt weitere Arten der Liste. Oben wurde die **einfach verkettete Liste** beschrieben, eine weitere ist die **doppelt verkettete Liste**.

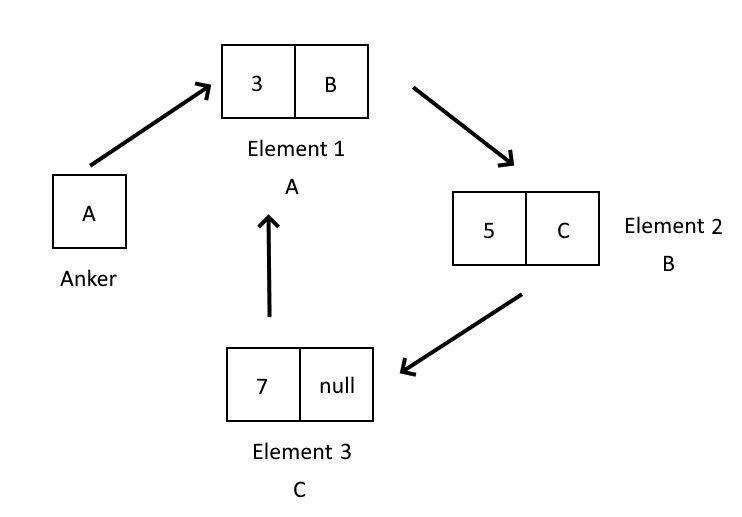


Dabei hat jedes Element nicht nur **einen** Zeiger, der auf das nächste Element zeigt, sondern zusätzlich noch einen Zeiger, der auf das **vorherige** Element zeigt.

Jetzt kann man zusätzlich die Liste von hinten nach vorne durchlaufen lassen.

### Kreisförmig verkettete Liste

Hier ist der Zeiger der Ende der Liste nicht **null**, sondern er zeigt auf das erste Element der Liste.



### Code

**List Deklaration**

List<int> list = new List<int>() { 4, 5 };

**Elemente anfügen**

list.Add(7);

**Elemente an Position x einfügen**

list.Insert(1, 3);

**Elemente an Position x Löschen**

list.RemoveAt(2);

**Index von Wert suchen**

list.IndexOf(3)

**Listgröße**

list.Count

**Liste leeren**

list.Clear();

**Code Beispiel**

List<int> list = new List<int>() { 4, 5 };

Console.WriteLine("Creating new list:");

Console.WriteLine(String.Join(" -> ", list) + "\n");

Console.WriteLine("Adding new number 7:");

list.Add(7);

Console.WriteLine(String.Join(" -> ", list) + "\n");

Console.WriteLine("Deleting number at index 2:");

list.RemoveAt(2);

Console.WriteLine(String.Join(" -> ", list) + "\n");

**Output**

Creating new list:

4 -> 5

Adding new number 7:

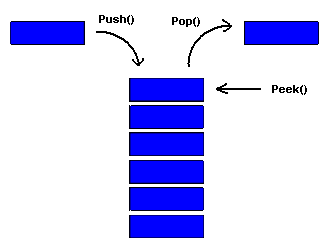
4 -> 5 -> 7

Deleting number at index 2:

4 -> 5

## Stacks[[5]](#footnote-5)

Stacks sind den Listen sehr ähnlich, funktionieren aber nach dem **LIFO** Prinzip, heißt **Last in First out**.



Visuell kann man sich den Stack wie einen Stapel vorstellen. Das was als letztes hinaufgelegt worden ist, muss auch als erstes wieder heruntergenommen werden.

Demnach gibt es zwei essenzielle Methoden des Stacks.

### Push

Fügt dem Stack ein Element hinzu.

### Pop

Nimmt das letzte Element weg.

### Code

C# hat zwei Arten von Stacks

**Stack<T>**

Jedes Element hat den gleichen Typ.

**Stack**

Die Elemente des Stacks sind vom Typ **object.**

**Stack Deklaration**

Stack<int> myStack = new Stack<int>();

**Alternativ mit Startwerten**

Stack<int> myStack = new Stack<int>(new int[] { 1, 2, 3, 4 });

**Elemente hinzufügen**

myStack.Push(1);

myStack.Push(3);

myStack.Push(2);

**Element löschen**

mystack.Pop();

**Hinweis**

Iteriert man über den Stack, so fängt man von hinten an.

**Code Beispiel**

Stack<int> myStack = new Stack<int>(new int[] { 2, 3, 1, 5, 4 });

foreach (int item in myStack)

{

Console.Write($"{item}, ");

}

Console.Writeline();

myStack.Pop();

Console.WriteLine(String.Join(", ", myStack));

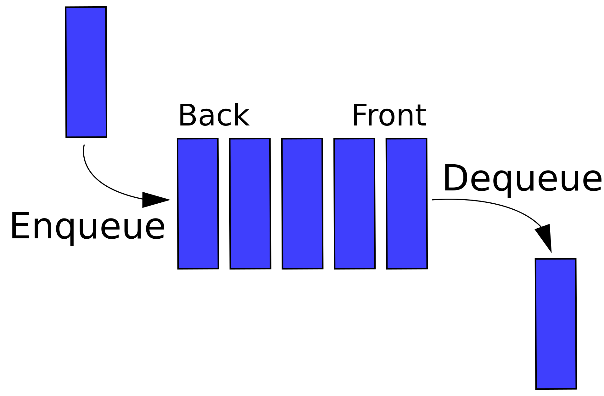
**Output**

4, 5, 1, 3, 2,

5, 1, 3, 2

## Queues[[6]](#footnote-6)

Queues sind den Listen sowie dem Stack sehr ähnlich, funktionieren aber nach dem **FIFO** Prinzip, heißt **First** **in First out**.



Visuell kann man sich die Queue wie einen Warteschlange vorstellen. Das was als erstes der Warteschlange hinzugefügt wurde worden ist, muss auch als erstes wieder heruntergenommen werden.

Demnach gibt es zwei essenzielle Methoden des Stacks.

### Enqueue

Fügt der Queue ein Element hinzu.

### Dequeue

Nimmt das erste Element weg.

### Code

**Code Beispiel**

Queue<int> myQueue = new Queue<int>();

myQueue.Enqueue(1);

myQueue.Enqueue(2);

myQueue.Enqueue(3);

myQueue.Enqueue(4);

Console.WriteLine(String.Join(", ", myQueue));

myQueue.Dequeue();

Console.WriteLine(String.Join(", ", myQueue));

**Output**

1, 2, 3, 4

2, 3, 4

## Hash Tables[[7]](#footnote-7)

Hashtables haben Key-Value Pairs. Dabei assoziiert man einen Key mit einem Value, sodass man den Value über den Key finden kann – „Key-Value Lookup“. Vorteil ist, dass der Lookup wesentlich schneller funktioniert, als würde man bspw. über eine Liste iterieren.

Beispiel:

Man hat eine Liste von Personen

„Tom“ => („Tom“, „Müller“, 28),

„Mark“ => („Mark“, „Gruber“, 21),

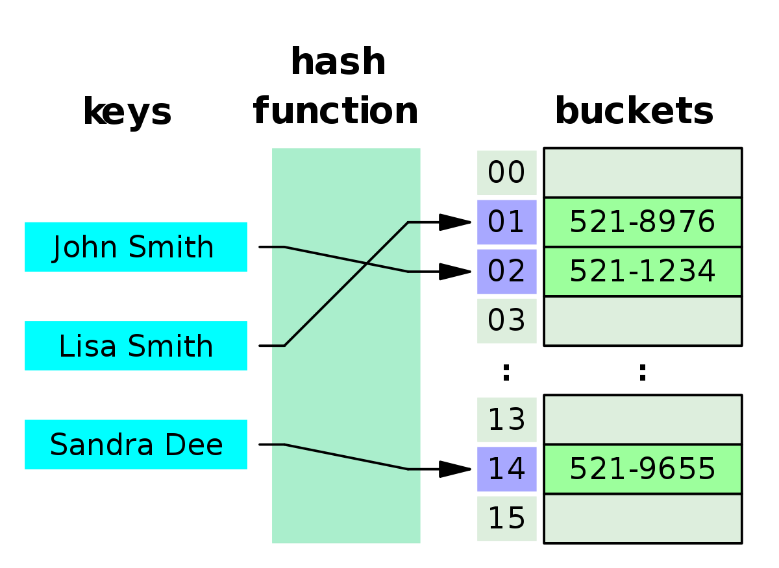
„Simon“ => („Simon“, „Maier“, 48)

Der Hashtable ermöglicht es die gewünschte Person in einer (fast) konstanten Zeit innerhalb der Liste zu finden.

Dazu braucht es zwei Sachen: Ein Array und einen Index. Heißt man muss „Tom“ zuerst in einen Index verwandeln. Dies funktioniert zum einen über die Hash-Funktion.

**Hash-Funktion**

Eine Hash-Funktion nimmt einen Input beliebiger länge und wandelt ihn in eine eindeutige Ausgabe mit fester Länge. Der Key muss hierbei nicht zwanghaft ein String sein, sondern kann eigentlich alles sein.

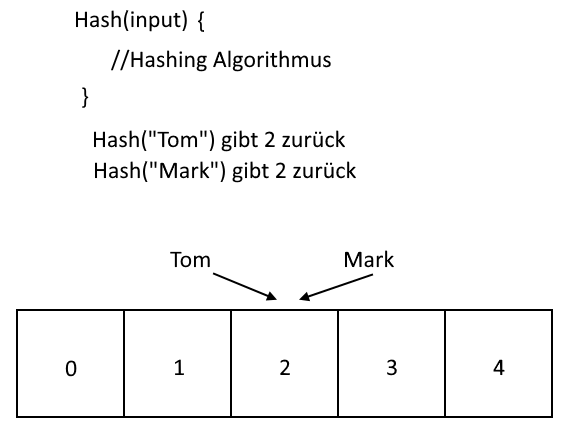


Dabei gibt es allerdings Milliarden von verschiedenen Möglichkeiten, die die Hashfunktion zurückgeben könnte.

Da das Array, was verwendet wird, nicht Milliarden von Feldern haben sollte, verwendet man wesentlich weniger Felder. Deshalb mappt man den Hash am Ende auf einen Index, der innerhalb des Arraybereiches liegt. Dabei können allerdings Kollisionen entstehen.

**Kollisionen**

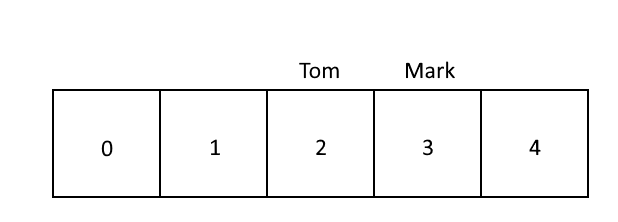
Kollisionen sind, wenn der gleiche Index bei verschiedenen Inputs generiert wird.



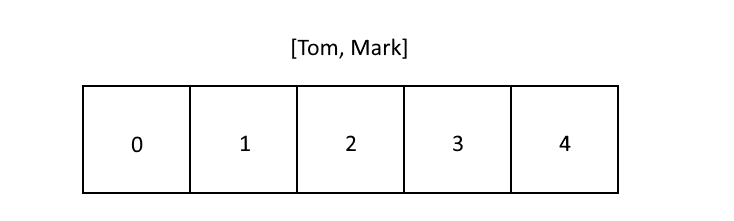
Ziel einer guten Hashfunktion ist es möglichst unabhängig vom Input einen möglichst gleichmäßige Aufteilung für die Indexe zu generieren, sodass möglichst wenige Kollisionen entstehen.

Mit diesen Kollisionen kann auf verschiedene Art und weise umgegangen werden.

1. Falls im Feld mit dem Index 2 bereits ein Wert ist, so verwendet man einfach das nächste freie Feld.



1. Man speichert Im Feld mit dem Index 2 eine Liste von Personen ab.



In beiden Varianten muss der Key mitgespeichert werden, um genau sagen zu können, welche der beiden Personen Objekte nun wirklich zu „Tom“ gehört.

### Code

In C# gibt es 2 Hauptvarianten von Hashtables

1. System.Collections.Hashtable

Non-Generic Type, objects können für Keys und Values verwendet werden.

1. System.Collections.Generic.Dictionary

Generic Type, die Typen aller Keys sind die gleichen und die Typen aller Values sind die gleichen.

**Code Beispiel**

Dictionary<string, Person> personen = new Dictionary<string, Person>();

personen.Add("Tom", new Person() {

vorname = "Tom",

nachname = "Müller",

alter = 28 });

personen.Add("Mark", new Person() {

vorname = "Mark",

nachname = "Gruber",

alter = 21 });

personen.Add("Simon", new Person() {

vorname = "Simon",

nachname = "Maier",

alter = 48});

Person gesucht = personen["Tom"];

Console.WriteLine($"{gesucht.vorname} {gesucht.nachname}, {gesucht.alter}");

class Person

{

public string vorname;

public string nachname;

public int alter;

}

**Output**

Tom Müller, 28

## Trees

## Heaps

## Graphs

# **ALGORITHMS**

# **REFERENZEN**

*Arrays.* (23. 04 2021). Von https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/arrays/ abgerufen

*Dynamic Arrays.* (30. 04 2021). Von https://www.geeksforgeeks.org/how-do-dynamic-arrays-work/ abgerufen

*Hash Tables.* (30. 04 2021). Von https://www.youtube.com/watch?v=shs0KM3wKv8 abgerufen

*Linked Lists (video).* (27. 04 2021). Von https://www.youtube.com/watch?v=x0k8MjvWNWw abgerufen

*Linked Lists.* (23. 04 2021). Von https://docs.microsoft.com/de-de/dotnet/api/system.collections.generic.list-1?view=net-5.0 abgerufen

*Queues.* (27. 04 2021). Von https://docs.microsoft.com/de-de/dotnet/api/system.collections.generic.queue-1?view=net-5.0 abgerufen

*Stacks.* (27. 04 2021). Von https://docs.microsoft.com/de-de/dotnet/api/system.collections.generic.stack-1?view=net-5.0 abgerufen

1. (Arrays, 2021) [↑](#footnote-ref-1)
2. (Dynamic Arrays, 2021) [↑](#footnote-ref-2)
3. (Linked Lists, 2021) [↑](#footnote-ref-3)
4. (Linked Lists (video), 2021) [↑](#footnote-ref-4)
5. (Stacks, 2021) [↑](#footnote-ref-5)
6. (Queues, 2021) [↑](#footnote-ref-6)
7. (Hash Tables, 2021) [↑](#footnote-ref-7)