Inhalt

Kapitel 1

Algorithmen und Datenstrukturen in ES6+

Dieses Buch ist noch in Bearbeitung....

1.1 Warum in ES6+

- ES is eating the world
    - meaningful understanding
    - I spent most of my professional life writing in JS and I think I know most about it

1.2 Behandelte Themen

- Algos und DS - Übungen am Ende. Versuche die Aufgaben zu machen ohne auf die Lösung zu schauen. Da ich finde, dass man von Fehlern oft viel mehr lernen kann als von richtigen Lösungen, habe ich vor der eigentlichen Lösung noch typische Fehler aufgelistet.

1.3 Nicht behandelte Themen

- Promises - async/await - webAPI - Browser spezifische Unterschiede

Kapitel 2

Entwicklungsumgebung

JS ist eine interpretierte Sprache. Sie läuft auf einer JS Engine. Die JS Engine läuft auf jeden Browser. Man kann direkt im Browser den Code ausführen.

Sobald man den Browser aber neu lädt ist unser Code weg. Besser ist es den Code entweder in nodejs auszuführen.

File speichern und mit node [filename] ausführen

Kapitel 3

JavaScript Paradigmen

JS ist eine sog. Multiparadigmen Programmiersprache. JS ist imperativ, objektorientiert und funktional.

3.1 Object-Orientiertes Programmieren

3.2 Funktionales Programmieren

Kapitel 4

JavaScript Konstrukte

4.1 Entscheidungskonstrukte

In JS gibt es if Statements, ternary und switch statement. if if else if else if ternary operator

Return

statement vs. expression switch

4.2 Wiederholungen

for loop, while loop, do while loop, symbol iterator; Generator; yields, for...of, for ...in

4.3 Funktionen

Es gibt verschiedene Arten Funktionen in JS zu definieren.

```
1 function functionDeclaration() {
2   //...
3 }
```

Listing 4.1: Array Konstruktor

```
1 var functionExpression = function() {
2   //...
3 }
```

Listing 4.2: Array Konstruktor

```
1 function() {
2   //...
3 }
```

Listing 4.3: Anonymous function

```
() => {
  //...
}
```

Listing 4.4: Arrow function

```
(function() {
  //..
})();
```

Listing 4.5: IIFE

```
var functionConstructor = new Function('a','b','return a + b');
```

Listing 4.6: IIFE

4.4 Scope

4.5 Closures

4.6 Hoisting

4.7 Call by Reference vs. Call by Value

4.8 Truthy und falsy

4.9 this

4.10 strict/sloppy mode

4.11 Number.EPSILON

Vergleich, ob Addition funktioniert

Kapitel 5

Datenstrukturen

Datenstrukturen lassen sich grob in drei Kategorien einteilen:

1. Array ähnliche Strukturen

   Dazu gehören Stacks und Queues. Diese Strukturen unterscheiden sich nur darin wie die Elemente eingefügt und entfernt werden.

2. Strukturen mit Knoten-Referenzen

   Strukturen, die eine Referenz zu einen Knoten haben sind LinkedList, Bäume und Graphen

3. Hash Tabellen Hash Tabellen sind von Hash Funktionen abhängig, um Daten zu speichern und zu finden.

5.1 Array

JS hat im Vergleich zu Java oder C/C++ nur sehr wenige Datenstrukturen. Eines ihrer wichtigsten Datenstrukturen ist der Array. Das Array werden wir später dazu benutzen, um alle anderen komplexeren Datenstrukturen zu implementieren. Der Unterschied zu JSs Arrays im Vergleich zu anderen Programmiersprachen ist, dass Arrays in JS keine fixe Länge haben. Durch das Hinzufügen und Entfernen von Elementen verändert sich die Array-Länge dynamisch mit. Bei der Initialisierung muss man dem Array dadurch auch keine bestimmte Länge mitgeben werden.

Arrays können in JS auf zwei Arten erstellt werden: Mit dem Array Konstruktor oder mit Array Literal:

```
1 var myArrayConstructor = new Array();
2 var myArrayLiteral = [];
```

Listing 5.1: Array Konstruktor

Der Array Konstruktur wird mit `new` eingeleitet und darauf folgt `Array()`. Beim Array Literal wird nur eine eckige Klammer `[]` benötigt. Beide Möglichkeiten

erstellen einen Array. Jedoch wird angeraten zur Erstellung eines Arrays das Array Literal zu nehmen. Nicht nur ist er kürzer und auch schneller, er ist auch syntaktisch eindeutiger. Denn mit dem Array Konstruktur kann man auch die Länge des Arrays definieren als auch initialisieren. Die Syntax von beiden Konstrukten sind sich ähnlich, sodass es zu Verwirrung kommen kann, wenn man den Array mit Zahlen initialisiert:

```
1 var myArrayLength = new Array(3);
2 var myArrayInit = new Array(3,2,1);
```

Listing 5.2: Array Konstruktor

Die Länge eines Arrays wird als Zahl (hier: 3) in den Konstruktur reingeschrieben. Damit hat das Array in unserem Beispiel eine Länge von 3, die man mit `length` überprüfen kann. Die (hier 3) Elemente sind noch `undefined`, da sie noch nicht initialisiert sind. Bei der Initialisierung gibt man die Elemente ebenfalls in den Konstruktur mit ein, jeweils getrennt durch einen Komma.

```
1  var myArrayLength = new Array(3);
2  console.log(myArrayLength);
3  // [undefined, undefined, undefined]
4  console.log(myArrayLenth.length);
5  // 3
6
7  var myArrayInit = new Array(3,2,1);
8  console.log(myArrayInit);
9  // [3, 2, 1]
10 console.log(myArrayInit.length);
11 // 3
```

Listing 5.3: Array Konstruktor

JS ist nicht static typed. D.h. ein Array kann Elemente nicht nur eines Typen gleichzeitig aufnehmen, sondern auch verschiedene. Ein Array in JS kann damit auch Zahlen, Strings, Bool und Objekte gleichzeitig aufnehmen:

```
1 var myArray = [1, "42", true, "hi", {"hello": "world"}];
```

Listing 5.4: Array Konstruktor

Intern werden die Elemente in einen String gecastet. Dadurch sind Arrays in JS langsamer als in anderen Sprachen. Um auf ein Array-Element zuzugreifen benutzen wir die eckige Klammer `[]`. Ein Array ist Index basiert und fängt mit `0` an. Um auf das zweite Element in einen Array zuzugreifen, schreiben wir also `myArray[1]`:

```
1 var myArray = [1, "42", true, "hi", {"hello": "world"}];
2 console.log(myArray[1]);
3 // "42"
```

Listing 5.5: Array Konstruktor

JS bieten viele Funktionen zur Manipulationen von Arrays an. Zum Hinzufügen am Ende wird `push()` benutzt. Um ein Element am Anfang des Arrays hinzuzufügen, wird `unshift()` verwendet:

```
var myArray = [1, 2, 3];
myArray.push(4);
console.log(myArray);
// [1, 2, 3, 4]

myArray.unshift(0);
console.log(myArray);
// [0, 1, 2, 3, 4]
```

Listing 5.6: Array Konstruktor

Für das Entfernen am Ende des Arrays gibt es `pop()`. Die Funktion `pop()` entfernt das letzte Element und gibt das entfernte Element zurück. Für das Entfernen am Anfang des Arrays verwendet man `shift()`.

```
var myArray = [0, 1, 2, 3, 4];
console.log(myArray.pop());
// 4
console.log(myArray);
// [0, 1, 2, 3]

console.log(myArray.shift());
// 0
console.log(myArray);
// [1, 2, 3]
```

Listing 5.7: Array Konstruktor

Um Elemente hinzuzufügen, zu ersetzen oder zu entfernen, die sich in der Mitte des Arrays befinden, verwendet man `splice()`. `splice()` nimmt als ersten Parameter den Index, an den das neue Element man hinzufügen will. Als zweiten Parameter wieviele Elemente danach ersetzt wird. Alle darauffolgenden Paramter die hinzuzufügenden Elemente.

```
var myArray = [1, 3, 4];
myArray.splice(1,0,2);
// adds a new element, 2, at index 1

myArray.splice(3,0,5,6,7);
// adds new elements, 5,6, and 7, at index 3

myArray.splice(5, 1);
// removes one element at index 5

myArray.splice(2, 3);
// removes 3 elements starting at index 2

myArray.splice(3, 1, 99);
// replaces one element at index 3 with the new element 99

myArray.splice(3, 2, 42);
// replaces 2 elements starting at index 3 with the new element 42

myArray.splice(999,0,2);
// if the index (first parameter) is larger than the array length,
    then it will just adds a new element to the end
```

Listing 5.8: Array Konstruktor

Gibt man nur eine Zahl als Argument mit, dann entfernt `splice()` die Elemente ab dem Index an der angegebenen Zahl bis zum Arrayende. Übergibt man eine Zahl, die gleich 0 ist oder größer oder gleich als die Länge des Arrays, dann wird das Ursprungsarray nicht verändert. Übergibt man eine negative Zahl, wird nicht von links, sondern von rechts gezählt und nach dem `n`-ten (von rechts) Element die Elemente entfernt.

Wenn nur ein Argument übergeben wird, dann gibt die `splice()` Funktion als Ergebnis die entfernten Elemente als Array zurück. Bei allen anderen Parameterübergaben, gibt sie nur einen leeren Array zurück.

```
1  var myArray = [1,2,3,4,5,6,7,8,9,10];
2  var removedElements = myArray.splice(4);
3  console.log(removedElements);
4  // [5,6,7,8,9,10]
5  console.log(myArray);
6  // [1,2,3,4]
7
8  var myArray = [1,2,3,4,5,6,7,8,9,10];
9  var removedElements = myArray.splice(99);
10 console.log(removedElements);
11 // []
12 console.log(myArray);
13 // array remains unchanged
14
15 var myArray = [1,2,3,4,5,6,7,8,9,10];
16 var removedElements = myArray.splice(-3);
17 console.log(removedElements);
18 // [8,9,10]
19 console.log(myArray);
20 // [1,2,3,4,5,6,7]
```

Listing 5.9: Array Konstruktor

Die Elemente in einen Array kann man mit `sort()` sortieren. `sort()` nimmt eine Methode auf, die zwei Elemente miteinander vergleicht und entweder eine positive oder negative Zahl zurückgibt oder `0`. Eine negative Zahl steht dafür, dass die Zahl kleiner ist. Eine positive Zahl steht dafür, dass die Zahl größer ist. Eine `0` steht dafür, dass beide Zahlen gleich sind.

Beim Sortieren von Strings ist keine Funktion notwendig. Jedoch kann man eine mitgeben bei der überprüft wird, ob ein String größer ist als ein anderer String. Wenn man jedoch bei der Sortierung von Zahlen sich auf die Default `sort()` Funktion verlässt, dann können die Zahlen falsch sortiert werden:

```
1  var myArray = [1, 5, 1001, 8, 4];
2  myArray.sort((a,b) => a - b);
3  console.log(myArray);
4  // [1, 4, 5, 8, 1001]
5
6  var myArray = [1, 5, 1001, 8, 4];
7  myArray.sort();
8  console.log(myArray);
9  // [1, 1001, 4, 5, 8]
10
11 var myArray = ["a", "c", "xxx", "bd"];
12 myArray.sort();
```

```
13 console.log(myArray);
14 // ["a", "bd", "c", "xxx"]
15
16 var myArray = ["a", "c", "xxx", "bd"];
17 myArray.sort((a, b) => a > b ? 1 : -1);
18 console.log(myArray);
19 // ["a", "bd", "c", "xxx"]
```

Listing 5.10: Array Konstruktor

Man kann die `sort()` Methode auch benutzen, um ein Array absteigend zu sortieren. Dazu kehrt man das Vorzeichen der mitzugebenden Funktion um:

```
1 var myArray = [1, 5, 1001, 8, 4];
2 myArray.sort((a,b) => b - a);
3 console.log(myArray);
4 // [1001, 8, 5, 4, 1]
5
6 var myArray = ["a", "c", "xxx", "bd"];
7 myArray.sort((a, b) => (a > b ? -1 : 1));
8 console.log(myArray);
9 // ["xxx", "c", "bd", "a"]
```

Listing 5.11: Array Konstruktor

Alternativ kann man zur Umkehrung des Arrays auch die `reverse()` Funktion benutzen. Diese ist sogar etwas schneller als nur die Sort Funktion mit der Methode und dem umgekehrten Vorzeichen von oben zu benutzen.

```
1 var myArray = [1, 5, 1001, 8, 4];
2 myArray.sort((a,b) => a - b).reverse();
3 console.log(myArray);
4 // [1001, 8, 5, 4, 1]
5
6 var myArray = ["a", "c", "xxx", "bd"];
7 myArray.sort().reverse();
8 console.log(myArray);
9 // ["xxx", "c", "bd", "a"]
```

Listing 5.12: Array Konstruktor

Die Funktionen `push()`, `unshift()`, `pop()`, `shift()`, `splice()`, usw. verändern den originären Array. Es gibt auch Funktionen, die das Original nicht ändert, sondern stattdessen einen neuen Array zurückliefert.

Die `map()` Methode geht durch die Elemente durch und wendet dabei eine ihr mitgegebene Methode `(currentValue, index, array)=> {}` auf jedes einzelne Element im Array an. Die ihr mitgegebene Methode nimmt als erstes Argument das aktuelle Element auf. Das zweite Element ist das momentane Index im Array. Das dritte Argument das ursprüngliche Array. Das Ergebnis ist wieder ein Array mit derselben Länge, wie das ursprüngliche Array:

```
1 var myArray = [1, 2, 3, 4];
2 function multiplyBy2 = item => item * 2;
3 var doubleArray = myArray.map(multiplyBy2);
4 console.log(doubleArray);
5 // [2, 4, 6, 8]
```

Listing 5.13: Array Konstruktor

Die `filter()` Methode nimmt ebenfalls eine Funktion als Argument auf und wendet sie auf alle Elemente im Array an. Nur wenn die Auswertung dieser Funktion auf das aktuelle Element `truthy` zurückgibt, wird dieses Element auch Teil des später zurückgegebenen Arrays. Die ihr übergebene Methode `(currentValue , index, array)=> {}` hat als erstes Argument das aktuelle Element. Das zweite Argument ist der aktuelle Index. Das dritte Argument das original Array auf das die `filter()` Methode angewendet wird. Nur das erste Argument ist verpflichtend. Die restlichen sind optional. Beispielhaft ist unten die `filter()` Methode zum Filtern von nur geraden Zahlen gezeigt:

```
1 var myArray = [1, 2, 3, 4, 5, 6];
2 function isEven = item => item % 2 ;
3 var onlyEven = myArray.map(isEven);
4 console.log(onlyEven);
5 // [2, 4, 6]
```

Listing 5.14: Array Konstruktor

Die `reduce()` Methode unterscheidet sich von `map()` und `filter()` dadurch, dass nicht immer ein Array zurück gegeben werden muss. Stattdessen reduziert die Methode das Array auf einen einzigen Wert. Dieser Wert kann eine Zahl oder auch ein Array sein. Als erstes Argument kann sie eine Methode nehmen. Als zweites Argument nimmt sie einen initial Wert an. Das zweite Argument ist nur optional. Die Methode, die sie aufnimmt, `(acc, item, index, array)=> {}` hat als erstes Argument einen Akkumulator, der den kumulierten Wert enthält und der am Ende das Ergebnis darstellt. Das zweite Argument ist der aktuelle Wert. Das dritte Argument der Index und das vierte das original Array. Nur die ersten beiden Argumente sind verpflichtend.

```
1 var myArray = [1, 2, 3];
2 var sum = (acc, item) => acc + item;
3 var sumOfArray = myArray.reduce(sum, 0);
4 // 6
```

Listing 5.15: Array Konstruktor

Die `reduce()` Funktion geht im Array von links nach rechts. Mit der `reduceRight ()` geht die Funktion von rechts nach links.

Die Methode `flat()`[1] wird auf Arrays angewendet, die selbst wiederum Arrays enthalten. Sie erstellt ein neues Arrays mit allen Unterarrays. Dabei kann optional bestimmt werden bis zu welcher Ebene die Unterarrays aufgelöst werden. Die `flat()` Methode nimmt optional nur einen numerischen Wert an. Dieser legt fest bis zu welcher Ebene die Unterarrays aufgelöst werden.

```
1 var myArray = [1, 2, 3, [4, 5, 6]];
2 console.log(myArray.flat());
3 // [1, 2, 3, 4, 5, 6]
4
5 var myArray = [1, 2, 3, [4, [5, 6]]];
6 console.log(myArray.flat());
7 // [1, 2, 3, 4, [5, 6]]
```

[1] noch im Experiment Status, d.h. es wurde noch nicht von allen JS Engines implementiert

```
8
9  var myArray = [1, 2, 3, [4, [5, 6]]];
10 console.log(myArray.flat(2));
11 // [1, 2, 3, 4, 5, 6]
```

Listing 5.16: Array Konstruktor

Die `flat()` Methode wird auch genutzt, um leere Elemente in Arrays zu entfernen

```
1 var myArray = [1, 2, 3, , ,6];
2 console.log(myArray.flat());
3 // [1, 2, 3, 6]
```

Listing 5.17: Array Konstruktor

Die Methode `flatMap()` [2] ist identisch zum Aufruf einer `map()` Methode gefolgt vom Aufruf einer `flat()` Methode. Die Methode wendet eine ihr mitgegebene Funktion auf alle Elemente an und flacht sie anschießend ab. Die ihr mitgegebene Funktion `(item, index, array)=> {}` nimmt als erstes Argument das aktuelle Element. Das zweite Argument der Index und das dritte Argument das Original Array.

```
1 var myArray = [1, 2, 3];
2 var duplicate = item => [item, item];
3 console.log(myArray.flatMap(duplicate));
4 // [1, 1, 2, 2, 3, 3]
```

Listing 5.18: Array Konstruktor

Die Funktion `concat()` vereint zwei Arrays und liefert das vereinigte Array als neues Array wieder zurück:

```
1 var myArray1 = ["a", "b", "c", "x"];
2 var myArray2 = ["d", "e", "f"];
3 var newArray = myArray1.concat(myArray2);
4 console.log(newArray);
5 // ["a", "b", "c", "x", "d", "e", "f"]
```

Listing 5.19: Array Konstruktor

Eine andere Möglichkeit ist es den Rest/Spread Operator zu verwenden:

```
1 var myArray1 = ["a", "b", "c", "x"];
2 var myArray2 = ["d", "e", "f"];
3 var newArray = [...myArray1, ...myArray2];
4 console.log(newArray);
5 // ["a", "b", "c", "x", "d", "e", "f"]
```

Listing 5.20: Array Konstruktor

Die `copyWithin()` Funktion kopiert einen Teil des Arrays in einer anderen Stelle desselben Arrays. Dabei wird die Länge des Arrays beibehalten.

```
1 var myArray1 = [1,2,3,4,5,6,7,8,9];
2 console.log(myArray1.copyWithin(1,3,6));
```

[2] noch im Experimentier Status, d.h. nicht alle Browser haben es implementiert

```
3 // Copies the numbers starting at index 3 to (exclusive) 6, i.e.
      [4,5,6] to the index 1
4 // [1,4,5,6,5,6,7,8,9]
5
6 console.log(myArray1.copyWithin(1,3));
7 // Copies the numbers starting at index 3 to the end, i.e.
      [4,5,6,7,8,9] to the index 1
8 // [1,4,5,6,7,8,9,8,9]
```

Listing 5.21: Array Konstruktor

Die `slice()` Methode erstellt eine (flache) Kopie des Arrays und gibt diese Kopie als neues Array zurück. Die Methode nimmt zwei Argumente auf. Das erste Argument beschreibt ab welchem Index die Kopie erstellt wird. Das zweite optionale Argument beschreibt bis zu welchem (exklusve) Index das Array kopiert wird. Wird für das zweite Argument kein Wert gegeben, dann kopiert er bis zum Arrayende:

```
1 var myArray = ["a", "b", "c", "x"];
2 var newArray = myArray.slice(0);
3 console.log(newArray);
4 // ["a", "b", "c", "x"]
```

Listing 5.22: Array Konstruktor

Eine flache Kopie deshalb, weil die Funktion die Elemente als Referenz in das neue Array schreibt. D.h. jede Änderung im original Array hat Auswirkung auf das neue Array. Dies gilt jedoch nicht für `Number`, `String`, `boolean`, `null`, `undefined`, `symbol`, sondern für `object`, `Array`, `function`:

```
1 // Using slice, create newCar from myCar.
2 var myHonda = { color: 'red', wheels: 4, engine: { cylinders: 4,
      size: 2.2 } };
3 var myCar = [myHonda, 2, 'cherry condition', 'purchased 1997'];
4 var newCar = myCar.slice(0, 2);
5 console.log(newCar);
6 // Display the values of myCar, newCar, and the color of myHonda
7 //  referenced from both arrays.
8 console.log('myCar[0].color = ' + myCar[0].color);
9 console.log('newCar[0].color = ' + newCar[0].color);
10
11 // Change the color of myHonda.
12 myHonda.color = 'purple';
13 console.log('The new color of my Honda is ' + myHonda.color);
14
15 // Display the color of myHonda referenced from both arrays.
16 console.log('myCar[0].color = ' + myCar[0].color);
17 console.log('newCar[0].color = ' + newCar[0].color);
```

Listing 5.23: Array Konstruktor

Eine weitere Möglichkeit den Array zu kopieren ist mit Hilfe des Rest/Spread-Operators:

```
1 var myArray = ["a", "b", "c", "x"];
2 var newArray = [...myArray];
3 console.log(newArray);
```

```
4 // ["a", "b", "c", "x"]
```

Listing 5.24: Array Konstruktor

Der Unterschied zwischen beiden Möglichkeiten ist, dass `slice()` eine flache Kopie und der Rest/Spread Operator eine tiefen Kopie erstellt:

```
1 var myArray = ["a", "b", "c", "x"];
2 var newArrayShallow = myArray.slice(0);
3 var newArrayDeep = [...myArray];
4 console.log(myArray === newArrayShallow);
5 // true --> it's pointing to the same memory space
6
7 console.log(myArray === newArrayDeep);
8 // false --> it's pointing to a new memory space
```

Listing 5.25: Array Konstruktor

Es sei hier erwähnt, dass der Rest/Spread Operator nur eine Ebene tief kopiert. Bei mehrdimensionalen Arrays muss man andere Methoden wählen. Unten sind Methoden aufgelistet mit der man eine tiefen Kopie, also einen Klon, erstellen kann:

```
1 var myArray = ["a", "b", "c", "x"];
2
3 var clonedArray1 = JSON.parse(JSON.stringify(myArray))
4 var clonedArray2 = [].concat(myArray);
5 var clonedArray2 = Array.from(myArray);
```

Listing 5.26: Array Konstruktor

Wie im vorherigen Kapitel angesprochen gibt es die Möglichkeit durch ein Array mit einer Schleife zu iterieren. Die Arrays in JS bieten eigene Funktionen zum Iterieren an. Die `forEach()` Funktion nimmt eine Methode und geht durch alle Elemente durch und wendet auf jedes einzelene ELement die Methode an. Die Funktion kann, im Gegensatz zu Schleifen wie `for`, `while`, `do while`, usw. nicht unterbrochen werden [3].Die `forEach()` Funktion liefert kein Ergebnis zurück.

```
1 [1,2,3,4,5].forEach((currentValue, index) => {
2   console.log(`Value ${currentValue} at index ${index}`);
3 });
```

Listing 5.27: Array Konstruktor

Will man vorher abbrechen so kann man entweder die `every()` oder `some()` verwenden. Die `every()` nimmt eine Funktion auf und überprüft sie für alle Elemente. Sobald bei der Überprüfung bei eines der Elemente `falsy` zurück gegeben wird, bricht sie ab. Bei der `some()` wird die Iteration abgebrochen sobald bei der Auswertung `truthy` zurückgegeben wird. Beide Funktionen geben `true` bzw. `false` zurück, wenn die Bedingung eingetroffen ist.

```
1 var hasLargerThanTen = [10,20,30].some(item => item > 10);
2 console.log(hasLargerThanTen);
3 // true
```

[3] außer durch das Werfen einer Ausnahme (was aber nicht empfohlen wird).

```
4
5  var hasSmallerThanTen = [10,20,30].some(item => item < 10);
6  console.log(hasSmallerThanTen);
7  // false
8
9  var allAreLargerThanTen = [10,20,30].every(item => item > 10);
10 console.log(allAreLargerThanTen);
11 // false
12
13 var allAreLargerThanTen = [100,20,30].every(item => item > 10);
14 console.log(allAreLargerThanTen);
15 // true
```

Listing 5.28: Array Konstruktor

Oft will man ein Element im Array finden. Dazu kann man wieder Schleifen benutzen oder die Built-in Funktionen verwenden.

Will man nur feststellen, ob ein Element auch im Array vorhanden ist, so kann man `includes()` verwenden. Dieses gibt entweder `true` oder `false` zurück, wenn das Element im Array gefunden bzw. nicht gefunden wurde.

```
1  var hasNumberTwo = [1,2,3,4,5].includes(2);
2  console.log(hasNumberTwo);
3  // true
4
5  var hasNumberTen = [1,2,3,4,5].includes(10);
6  console.log(hasNumberTen);
7  // false
```

Listing 5.29: Array Konstruktor

Will man noch dazu herauszufinden, an welcher Position das gesuchte Element sich befindet, gibt es `indexOf()`. Diese nimmmt eine Funktion auf und liefert den ersten Index des Elements, bei der die mitgegebene Funktion `truthy` zurückliefert. `lastIndexOf()` macht genau das Gegenteil: sie gibt den Index des zu letzt gefundenen Elements zurück. Falls keines der Elemente den Bedingungen entspricht, liefert `indexOf()` und `lastIndexOf()` `-1` zurück.

```
1  var position = ["a", "b", "c"]
```

Listing 5.30: Array Konstruktor

Will man einen `String`in einen Array umwandeln, so kann man wieder den Rest/Spread-Operator verwenden. Dabei wird jedes einzelne Zeichen, inklusive Leerzeichen, Komma, Sonderzeichen usw., als eigenes Element in ein neues Array gepackt und zurückgegeben.

```
1  var myString = "hello, world";
2  var myArray = [...myString];
3  console.log(myArray);
4  // ["h", "e", "l", "l", "o", ",", " ", "w", "o", "r", "l", "d"]
```

Listing 5.31: Array Konstruktor

Mit `split()` kann man festlegen ab wann man die Elemente in einen Array übergeben will. Die Funktion nimmt einen `String` als Prädikat auf. Will man

z.B. beim obigen Beispiel nur die Wörter in den Array geben, die durch einen Komma getrennt sind, so gibt man das als String in die Funktion ein:

```
1 var myString = "hello, world";
2 var myArray = myString.split(",")
3 console.log(myArray);
4 // ["hello", " world"]
```

Listing 5.32: Array Konstruktor

5.2 Liste

Listen sind gut geeignet, wenn es nicht auf die Reihenfolge der Elemente ankommt und wenn man nicht nach einen bestimmten Element suchen muss. Nachfolgend wollen wir eine `List`-Klasse implementieren, die folgende Eigenschaften und Funktionen hat:

- pos (property): Current position in list
- size (property): Returns the number of elements in list
- clear (function): Clears all elements from list
- toString (function): Returns string representation of list
- get(function): Returns element at specified index. If nothing is specified then return element at current position
- indexOf (function): Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element
- lastIndexOf (function): Returns the index of the last occurrence of the specified element in this list, or -1 if the list doesn't contain the specified element
- isEmpty (function): Returns true if the list contains no element
- insert (function): Inserts new element after existing element
- add (function): Appends new element to end of list
- remove (function): Removes element at a specific position from list
- front (function): Sets current position to first element of list
- end (function): Sets current position to last element of list
- prev (function): Moves current position back one element
- next (function): Moves current position forward one element
- moveTo (function): Moves the current position to specified position
- sublist(function): Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

List Klasse Nachfolgend werden wir die `List` Klasse erstellen. Die gesamte Klasse inklusive den Tests zu dieser Klasse kann unter meinem GitHub Repo angeschaut werden. Die ganze Klasse findet ihr hier: `https://github.com/pandaquests/Datastructure_Algorithm_ES6/blob/master/code/datastructures/List.js`

Der Test zu der Klasse ist hier: `https://github.com/pandaquests/Datastructure_Algorithm_ES6/blob/master/code/test/datastructures/List.test.js`

Der Konstruktor kann einen Array aufnehmen oder leer bleiben. Wenn er leer bleibt, dann wird eine leere Liste erstellt.

```
class List {
  constructor(items = []) {
    this.store = items;
    this.pos = 0;
    this.size = items.length;
  }
  clear() { /*...*/ }
  toString() {/*...*/ }
  get(index) {/*...*/ }
  indexOf (item) {/*...*/}
  lastIndexOf () {/*...*/}
  isEmpty () {/*...*/}
  insert(index) {/*...*/ }
  add() {/*...*/ }
  remove() {/*...*/ }
  front() {/*...*/ }
  end() {/*...*/ }
  prev() {/*...*/ }
  next() {/*...*/ }
  moveTo(index) {/*...*/ }
  sublist(from, to) {/*...*/}
}
```

Listing 5.33: Array Konstruktor

Die einzelenen noch leeren Funktionen werden wir nachfolgend mit Leben füllen. Die Eigenschaften können wir aber schon setzen. Bei einer Liste sind `listSize`, `pos`, `size` zu Anfang selbstverständlich alle `0`.

Erklären, was man mit der Position macht..... als Iterable..

clear() - Löschen aller Elemente einer Liste Beim Löschen muss der `dataStore` sowie die Eigenschaft Länge und Position wieder auf den Initialzustand zurück versetzt werden.

```
clear() {
  this.store = [];
  this.size = this.pos = 0;
}
```

Listing 5.34: Array Konstruktor

toString() - Gibt die Repräsentation der Liste als String zurück Bei der toString Funktion reicht es aus, wenn wir die `dataStore` zurückgeben.

```
toString() {
  return this.store;
}
```

Listing 5.35: Array Konstruktor

get(index) - Ein Element an einem bestimmten Index bekommen Diese Funktion bekommt einen Defaultwert. Falls der User nur `get()` aufruft, bekommt er das Element bei der `index = pos` ist. Gibt er eine Zahl ein, z.B. `get(2)` bekommt er das Element ber der Index dieser Zahl entspricht.

```
get(index = this.pos) {
  return this.store[index];
}
```

Listing 5.36: Array Konstruktor

indexOf(item) - Gibt den Index wo das gesuchte Element zuerst vorkommt Diese Funktion kann nur auf primitive Typen, wie `String`, `Number`, `Boolean`, usw. angewandt werden. Wir verwenden dazu die native `indexOf()` Funktion von JS.

```
indexOf(item) {
  return this.store.indexOf(item);
}
```

Listing 5.37: Array Konstruktor

lastIndexOf(item) - Gibt den Index wo das gesuchte Element zuletzt vorkommt Auch diese Funktion kann nur auf primitive Typen, wie `String`, `Number`, `Boolean`, usw. angewandt werden. Wir verwenden dazu die native `indexOf()` Funktion von JS.

```
lastIndexOf(item) {
  return this.store.lastIndexOf(item);
}
```

Listing 5.38: Array Konstruktor

```
isEmpty() {
  return this.store.length === 0;
}
```

Listing 5.39: Array Konstruktor

Hier vergleichen wir die Größe des Arrays mit der Zahl 0. Alternativ hätten wir auch `return Boolean(this.store.length)` schreiben können. Jedoch finde ich die obige Version deklarativer. Dies ist manchmal viel wichtiger als kurzer oder prägnanter Code, weil wir den Code nicht für die Maschine schreiben, sondern für andere Entwickler.

```
insert(el, index) {
  this.store.splice(index, 0, el);
}
```

Listing 5.40: Array Konstruktor

```
add(el) {
  this.store.push(el);
}
```

Listing 5.41: Array Konstruktor

```
remove(index) {
  this.store.splice(index, 1);
}
```

Listing 5.42: Array Konstruktor

```
remove(index) {
  this.pos = 0;
}
```

Listing 5.43: Array Konstruktor

```
end() {
  this.pos = this.store.length - 1;
}
```

Listing 5.44: Array Konstruktor

```
prev() {
  if (this.pos === 0) {
    return 0;
  }
  return --this.pos;
}
```

Listing 5.45: Array Konstruktor

```
next() {
  if (this.pos === this.store.length - 1) {
    return this.pos;
  }
  return ++this.pos;
}
```

Listing 5.46: Array Konstruktor

```
1 moveTo(index) {
2   if (this.pos === this.store.length - 1
3     || this.pos === 0) {
4     return;
5   }
6   this.pos = index;
7 }
```

Listing 5.47: Array Konstruktor

sublist(from, to) - Erstellt eine Liste von (inklusive) einer bestimmten Stelle bis (exklusiv) eine bestimmten Stelle ohne das original Array zu ändern Die Funktion `slice()` bietet sich dafür besonders gut an, da sie genau eine subliste nach genau den genannten Parametern erstellt und dabei das original Array nicht verändert.

```
1 sublist(from, to) {
2   return this.store.slice(from, to);
3 }
```

Listing 5.48: Array Konstruktor

5.3 Stack

Ein Stack ist wie ein Stapel von Papier. Man legt ein Blatt Papier oben auf den Stapel auf und wenn man eins wegnehmen will, dann kann man dieses nur von oben entfernen. Dieses Prinzip nennt sich "Last-in-first-out" (LIFO). Viele Funktionen des `Stacks` werden genauso implementiert wie bei der `List` . Daher werde ich hier nur die Funktonen beschreiben, die ich bisher noch nicht implementiert habe. Für die komplette Implementierung siehe mein Repo inklusive Test.

Der Funktionsumfang eines `Stack`, schaut folgendermaßen aus:

- push(item): Current position in list
- pop (property): Returns the number of elements in list
- peek (function): Clears all elements from list
- empty (function): Returns string representation of list
- search(el)

push public E push(E item) Pushes an item onto the top of this stack. This has exactly the same effect as: addElement(item) Parameters: item - the item to be pushed onto this stack. Returns: the item argument. See Also: Vector.addElement(E) pop public E pop() Removes the object at the top of this stack and returns that object as the value of this function. Returns: The object at the top of this stack (the last item of the Vector object). Throws:

EmptyStackException - if this stack is empty. peek public E peek() Looks at the object at the top of this stack without removing it from the stack. Returns: the object at the top of this stack (the last item of the Vector object). Throws: EmptyStackException - if this stack is empty. empty public boolean empty() Tests if this stack is empty. Returns: true if and only if this stack contains no items; false otherwise. search public int search(Object o) Returns the 1-based position where an object is on this stack. If the object o occurs as an item in this stack, this method returns the distance from the top of the stack of the occurrence nearest the top of the stack; the topmost item on the stack is considered to be at distance 1. The equals method is used to compare o to the items in this stack. Parameters: o - the desired object. Returns: the 1-based position from the top of the stack where the object is located; the return value -1 indicates that the object is not on the stack.

5.4 Queue

- add boolean add(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available. Specified by: add in interface Collection¡E¿ Parameters: e - the element to add Returns: true (as specified by Collection.add(E)) Throws: IllegalStateException - if the element cannot be added at this time due to capacity restrictions ClassCastException - if the class of the specified element prevents it from being added to this queue NullPointerException - if the specified element is null and this queue does not permit null elements IllegalArgumentException - if some property of this element prevents it from being added to this queue

- offer boolean offer(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted queue, this method is generally preferable to add(E), which can fail to insert an element only by throwing an exception. Parameters: e - the element to add Returns: true if the element was added to this queue, else false Throws: ClassCastException - if the class of the specified element prevents it from being added to this queue NullPointerException - if the specified element is null and this queue does not permit null elements IllegalArgumentException - if some property of this element prevents it from being added to this queue

- remove E remove() Retrieves and removes the head of this queue. This method differs from poll only in that it throws an exception if this queue is empty. Returns: the head of this queue Throws: NoSuchElementException - if this queue is empty

- poll E poll() Retrieves and removes the head of this queue, or returns null if this queue is empty. Returns: the head of this queue, or null if this

queue is empty element E element() Retrieves, but does not remove, the head of this queue. This method differs from peek only in that it throws an exception if this queue is empty. Returns: the head of this queue Throws: NoSuchElementException - if this queue is empty

- addFirst void addFirst(E e) Inserts the specified element at the front of this deque if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted deque, it is generally preferable to use method offerFirst(E). Parameters: e - the element to add Throws: IllegalStateException - if the element cannot be added at this time due to capacity restrictions ClassCastException - if the class of the specified element prevents it from being added to this deque NullPointerException - if the specified element is null and this deque does not permit null elements IllegalArgumentException - if some property of the specified element prevents it from being added to this deque

- peek E peek() Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. Returns: the head of this queue, or null if this queue is empty

Method Detail

5.5 Dequeue

- addLast void addLast(E e) Inserts the specified element at the end of this deque if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted deque, it is generally preferable to use method offerLast(E). This method is equivalent to add(E).

  Parameters: e - the element to add Throws: IllegalStateException - if the element cannot be added at this time due to capacity restrictions ClassCastException - if the class of the specified element prevents it from being added to this deque NullPointerException - if the specified element is null and this deque does not permit null elements IllegalArgumentException - if some property of the specified element prevents it from being added to this deque

- offerFirst boolean offerFirst(E e) Inserts the specified element at the front of this deque unless it would violate capacity restrictions. When using a capacity-restricted deque, this method is generally preferable to the addFirst(E) method, which can fail to insert an element only by throwing an exception. Parameters: e - the element to add Returns: true if the element was added to this deque, else false Throws: ClassCastException - if the class of the specified element prevents it from being added to this deque NullPointerException - if the specified element is null and this deque does not permit null elements IllegalArgumentException - if some property of the specified element prevents it from being added to this deque

- offerLast boolean offerLast(E e) Inserts the specified element at the end of this deque unless it would violate capacity restrictions. When using a capacity-restricted deque, this method is generally preferable to the addLast(E) method, which can fail to insert an element only by throwing an exception. Parameters: e - the element to add Returns: true if the element was added to this deque, else false Throws: ClassCastException - if the class of the specified element prevents it from being added to this deque NullPointerException - if the specified element is null and this deque does not permit null elements IllegalArgumentException - if some property of the specified element prevents it from being added to this deque

- removeFirst E removeFirst() Retrieves and removes the first element of this deque. This method differs from pollFirst only in that it throws an exception if this deque is empty. Returns: the head of this deque Throws: NoSuchElementException - if this deque is empty

- removeLast E removeLast() Retrieves and removes the last element of this deque. This method differs from pollLast only in that it throws an exception if this deque is empty. Returns: the tail of this deque Throws: NoSuchElementException - if this deque is empty pollFirst E pollFirst() Retrieves and removes the first element of this deque, or returns null if this deque is empty. Returns: the head of this deque, or null if this deque is empty

- pollLast E pollLast() Retrieves and removes the last element of this deque, or returns null if this deque is empty. Returns: the tail of this deque, or null if this deque is empty

- getFirst E getFirst() Retrieves, but does not remove, the first element of this deque. This method differs from peekFirst only in that it throws an exception if this deque is empty. Returns: the head of this deque Throws: NoSuchElementException - if this deque is empty

- getLast E getLast() Retrieves, but does not remove, the last element of this deque. This method differs from peekLast only in that it throws an exception if this deque is empty. Returns: the tail of this deque Throws: NoSuchElementException - if this deque is empty

- peekFirst E peekFirst() Retrieves, but does not remove, the first element of this deque, or returns null if this deque is empty. Returns: the head of this deque, or null if this deque is empty

- peekLast E peekLast() Retrieves, but does not remove, the last element of this deque, or returns null if this deque is empty. Returns: the tail of this deque, or null if this deque is empty

- removeFirstOccurrence boolean removeFirstOccurrence(Object o) Removes the first occurrence of the specified element from this deque. If the deque

does not contain the element, it is unchanged. More formally, removes the first element e such that (o==null ? e==null : o.equals(e)) (if such an element exists). Returns true if this deque contained the specified element (or equivalently, if this deque changed as a result of the call). Parameters: o - element to be removed from this deque, if present Returns: true if an element was removed as a result of this call Throws: ClassCastException - if the class of the specified element is incompatible with this deque (optional) NullPointerException - if the specified element is null and this deque does not permit null elements (optional)

- removeLastOccurrence boolean removeLastOccurrence(Object o) Removes the last occurrence of the specified element from this deque. If the deque does not contain the element, it is unchanged. More formally, removes the last element e such that (o==null ? e==null : o.equals(e)) (if such an element exists). Returns true if this deque contained the specified element (or equivalently, if this deque changed as a result of the call). Parameters: o - element to be removed from this deque, if present Returns: true if an element was removed as a result of this call Throws: ClassCastException - if the class of the specified element is incompatible with this deque (optional) NullPointerException - if the specified element is null and this deque does not permit null elements (optional)

- add boolean add(E e) Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque) if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available. When using a capacity-restricted deque, it is generally preferable to use offer. This method is equivalent to addLast(E).

  Specified by: add in interface Collection¡E¿ Specified by: add in interface Queue¡E¿ Parameters: e - the element to add Returns: true (as specified by Collection.add(E)) Throws: IllegalStateException - if the element cannot be added at this time due to capacity restrictions ClassCastException - if the class of the specified element prevents it from being added to this deque NullPointerException - if the specified element is null and this deque does not permit null elements IllegalArgumentException - if some property of the specified element prevents it from being added to this deque

- offer boolean offer(E e) Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque) if it is possible to do so immediately without violating capacity restrictions, returning true upon success and false if no space is currently available. When using a capacity-restricted deque, this method is generally preferable to the add(E) method, which can fail to insert an element only by throwing an exception. This method is equivalent to offerLast(E).

  Specified by: offer in interface Queue¡E¿ Parameters: e - the element to add Returns: true if the element was added to this deque, else false

Throws: ClassCastException - if the class of the specified element prevents it from being added to this deque NullPointerException - if the specified element is null and this deque does not permit null elements IllegalArgumentException - if some property of the specified element prevents it from being added to this deque

- remove E remove() Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque). This method differs from poll only in that it throws an exception if this deque is empty. This method is equivalent to removeFirst().

  Specified by: remove in interface Queue¡E¿ Returns: the head of the queue represented by this deque Throws: NoSuchElementException - if this deque is empty

- poll E poll() Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty. This method is equivalent to pollFirst().

  Specified by: poll in interface Queue¡E¿ Returns: the first element of this deque, or null if this deque is empty element E element() Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque). This method differs from peek only in that it throws an exception if this deque is empty. This method is equivalent to getFirst().

  Specified by: element in interface Queue¡E¿ Returns: the head of the queue represented by this deque Throws: NoSuchElementException - if this deque is empty

- peek E peek() Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty. This method is equivalent to peekFirst().

  Specified by: peek in interface Queue¡E¿ Returns: the head of the queue represented by this deque, or null if this deque is empty

- push void push(E e) Pushes an element onto the stack represented by this deque (in other words, at the head of this deque) if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available. This method is equivalent to addFirst(E).

  Parameters: e - the element to push Throws: IllegalStateException - if the element cannot be added at this time due to capacity restrictions ClassCastException - if the class of the specified element prevents it from being added to this deque NullPointerException - if the specified element is null and this deque does not permit null elements IllegalArgumentException - if some property of the specified element prevents it from being added to this deque

- pop E pop() Pops an element from the stack represented by this deque. In other words, removes and returns the first element of this deque. This method is equivalent to removeFirst().

  Returns: the element at the front of this deque (which is the top of the stack represented by this deque) Throws: NoSuchElementException - if this deque is empty

- remove boolean remove(Object o) Removes the first occurrence of the specified element from this deque. If the deque does not contain the element, it is unchanged. More formally, removes the first element e such that (o==null ? e==null : o.equals(e)) (if such an element exists). Returns true if this deque contained the specified element (or equivalently, if this deque changed as a result of the call). This method is equivalent to removeFirstOccurrence(java.lang.Object).

  Specified by: remove in interface Collection¡E¿ Parameters: o - element to be removed from this deque, if present Returns: true if an element was removed as a result of this call Throws: ClassCastException - if the class of the specified element is incompatible with this deque (optional) NullPointerException - if the specified element is null and this deque does not permit null elements (optional)

- contains boolean contains(Object o) Returns true if this deque contains the specified element. More formally, returns true if and only if this deque contains at least one element e such that (o==null ? e==null : o.equals(e)). Specified by: contains in interface Collection¡E¿ Parameters: o - element whose presence in this deque is to be tested Returns: true if this deque contains the specified element Throws: ClassCastException - if the type of the specified element is incompatible with this deque (optional) NullPointerException - if the specified element is null and this deque does not permit null elements (optional)

- size int size() Returns the number of elements in this deque. Specified by: size in interface Collection¡E¿ Returns: the number of elements in this deque

- iterator Iterator¡E¿ iterator() Returns an iterator over the elements in this deque in proper sequence. The elements will be returned in order from first (head) to last (tail). Specified by: iterator in interface Collection¡E¿ Specified by: iterator in interface Iterable¡E¿ Returns: an iterator over the elements in this deque in proper sequence

- descendingIterator Iterator¡E¿ descendingIterator() Returns an iterator over the elements in this deque in reverse sequential order. The elements will be returned in order from last (tail) to first (head). Returns: an iterator over the elements in this deque in reverse sequence

Method Detail

5.6 Priority Queue

Method Detail add public boolean add(E e) Inserts the specified element into this priority queue. Specified by: add in interface Collection¡E¿ Specified by: add in interface Queue¡E¿ Overrides: add in class AbstractQueue¡E¿ Parameters: e - the element to add Returns: true (as specified by Collection.add(E)) Throws: ClassCastException - if the specified element cannot be compared with elements currently in this priority queue according to the priority queue's ordering NullPointerException - if the specified element is null offer public boolean offer(E e) Inserts the specified element into this priority queue. Specified by: offer in interface Queue¡E¿ Parameters: e - the element to add Returns: true (as specified by Queue.offer(E)) Throws: ClassCastException - if the specified element cannot be compared with elements currently in this priority queue according to the priority queue's ordering NullPointerException - if the specified element is null peek public E peek() Description copied from interface: Queue Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. Specified by: peek in interface Queue¡E¿ Returns: the head of this queue, or null if this queue is empty remove public boolean remove(Object o) Removes a single instance of the specified element from this queue, if it is present. More formally, removes an element e such that o.equals(e), if this queue contains one or more such elements. Returns true if and only if this queue contained the specified element (or equivalently, if this queue changed as a result of the call). Specified by: remove in interface Collection¡E¿ Overrides: remove in class AbstractCollection¡E¿ Parameters: o - element to be removed from this queue, if present Returns: true if this queue changed as a result of the call contains public boolean contains(Object o) Returns true if this queue contains the specified element. More formally, returns true if and only if this queue contains at least one element e such that o.equals(e). Specified by: contains in interface Collection¡E¿ Overrides: contains in class AbstractCollection¡E¿ Parameters: o - object to be checked for containment in this queue Returns: true if this queue contains the specified element toArray public Object[] toArray() Returns an array containing all of the elements in this queue. The elements are in no particular order. The returned array will be "safe" in that no references to it are maintained by this queue. (In other words, this method must allocate a new array). The caller is thus free to modify the returned array.

This method acts as bridge between array-based and collection-based APIs.

Specified by: toArray in interface Collection¡E¿ Overrides: toArray in class AbstractCollection¡E¿ Returns: an array containing all of the elements in this queue toArray public ¡T¿ T[] toArray(T[] a) Returns an array containing all of the elements in this queue; the runtime type of the returned array is that of the specified array. The returned array elements are in no particular order. If the queue fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this queue. If the queue fits in the specified array with room to spare (i.e., the array has more elements than the queue), the element in the array immediately following the end of the collection is set to null.

Like the toArray() method, this method acts as bridge between array-based and collection-based APIs. Further, this method allows precise control over the runtime type of the output array, and may, under certain circumstances, be used to save allocation costs.

Suppose x is a queue known to contain only strings. The following code can be used to dump the queue into a newly allocated array of String:

String[] y = x.toArray(new String[0]); Note that toArray(new Object[0]) is identical in function to toArray(). Specified by: toArray in interface Collection¡E¿ Overrides: toArray in class AbstractCollection¡E¿ Parameters: a - the array into which the elements of the queue are to be stored, if it is big enough; otherwise, a new array of the same runtime type is allocated for this purpose. Returns: an array containing all of the elements in this queue Throws: ArrayStoreException - if the runtime type of the specified array is not a supertype of the runtime type of every element in this queue NullPointerException - if the specified array is null iterator public Iterator¡E¿ iterator() Returns an iterator over the elements in this queue. The iterator does not return the elements in any particular order. Specified by: iterator in interface Iterable¡E¿ Specified by: iterator in interface Collection¡E¿ Specified by: iterator in class AbstractCollection¡E¿ Returns: an iterator over the elements in this queue size public int size() Description copied from interface: Collection Specified by: size in interface Collection¡E¿ Specified by: size in class AbstractCollection¡E¿ Returns: the number of elements in this collection clear public void clear() Removes all of the elements from this priority queue. The queue will be empty after this call returns. Specified by: clear in interface Collection¡E¿ Overrides: clear in class AbstractQueue¡E¿ poll public E poll() Description copied from interface: Queue Retrieves and removes the head of this queue, or returns null if this queue is empty. Specified by: poll in interface Queue¡E¿ Returns: the head of this queue, or null if this queue is empty comparator public Comparator¡? super E¿ comparator() Returns the comparator used to order the elements in this queue, or null if this queue is sorted according to the natural ordering of its elements. Returns: the comparator used to order this queue, or null if this queue is sorted according to the natural ordering of its elements

5.7 LinkedList

Method Detail getFirst public E getFirst() Returns the first element in this list. Specified by: getFirst in interface Deque¡E¿ Returns: the first element in this list Throws: NoSuchElementException - if this list is empty getLast public E getLast() Returns the last element in this list. Specified by: getLast in interface Deque¡E¿ Returns: the last element in this list Throws: NoSuchElementException - if this list is empty removeFirst public E removeFirst() Removes and returns the first element from this list. Specified by: removeFirst in interface Deque¡E¿ Returns: the first element from this list Throws: NoSuchElementException - if this list is empty removeLast public E removeLast() Removes and returns the last element from this list. Specified by: removeLast in interface

Deque¡E¿ Returns: the last element from this list Throws: NoSuchElementException - if this list is empty addFirst public void addFirst(E e) Inserts the specified element at the beginning of this list. Specified by: addFirst in interface Deque¡E¿ Parameters: e - the element to add addLast public void addLast(E e) Appends the specified element to the end of this list. This method is equivalent to add(E).

Specified by: addLast in interface Deque¡E¿ Parameters: e - the element to add contains public boolean contains(Object o) Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)). Specified by: contains in interface Collection¡E¿ Specified by: contains in interface Deque¡E¿ Specified by: contains in interface List¡E¿ Overrides: contains in class AbstractCollection¡E¿ Parameters: o - element whose presence in this list is to be tested Returns: true if this list contains the specified element size public int size() Returns the number of elements in this list. Specified by: size in interface Collection¡E¿ Specified by: size in interface Deque¡E¿ Specified by: size in interface List¡E¿ Specified by: size in class AbstractCollection¡E¿ Returns: the number of elements in this list add public boolean add(E e) Appends the specified element to the end of this list. This method is equivalent to addLast(E).

Specified by: add in interface Collection¡E¿ Specified by: add in interface Deque¡E¿ Specified by: add in interface List¡E¿ Specified by: add in interface Queue¡E¿ Overrides: add in class AbstractList¡E¿ Parameters: e - element to be appended to this list Returns: true (as specified by Collection.add(E)) remove public boolean remove(Object o) Removes the first occurrence of the specified element from this list, if it is present. If this list does not contain the element, it is unchanged. More formally, removes the element with the lowest index i such that (o==null ? get(i)==null : o.equals(get(i))) (if such an element exists). Returns true if this list contained the specified element (or equivalently, if this list changed as a result of the call). Specified by: remove in interface Collection¡E¿ Specified by: remove in interface Deque¡E¿ Specified by: remove in interface List¡E¿ Overrides: remove in class AbstractCollection¡E¿ Parameters: o - element to be removed from this list, if present Returns: true if this list contained the specified element addAll public boolean addAll(Collection¡? extends E¿ c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. The behavior of this operation is undefined if the specified collection is modified while the operation is in progress. (Note that this will occur if the specified collection is this list, and it's nonempty.) Specified by: addAll in interface Collection¡E¿ Specified by: addAll in interface List¡E¿ Overrides: addAll in class AbstractCollection¡E¿ Parameters: c - collection containing elements to be added to this list Returns: true if this list changed as a result of the call Throws: NullPointerException - if the specified collection is null See Also: AbstractCollection.add(Object) addAll public boolean addAll(int index, Collection¡? extends E¿ c) Inserts all of the elements in the specified collection into this list, starting at the specified position. Shifts the element currently at that position (if any) and any subsequent elements to the right (increases their

indices). The new elements will appear in the list in the order that they are returned by the specified collection's iterator. Specified by: addAll in interface List¡E¿ Overrides: addAll in class AbstractSequentialList¡E¿ Parameters: index - index at which to insert the first element from the specified collection c - collection containing elements to be added to this list Returns: true if this list changed as a result of the call Throws: IndexOutOfBoundsException - if the index is out of range (index ¡ 0 —— index ¿ size()) NullPointerException - if the specified collection is null clear public void clear() Removes all of the elements from this list. The list will be empty after this call returns. Specified by: clear in interface Collection¡E¿ Specified by: clear in interface List¡E¿ Overrides: clear in class AbstractList¡E¿ get public E get(int index) Returns the element at the specified position in this list. Specified by: get in interface List¡E¿ Overrides: get in class AbstractSequentialList¡E¿ Parameters: index - index of the element to return Returns: the element at the specified position in this list Throws: IndexOutOfBoundsException - if the index is out of range (index ¡ 0 —— index ¿= size()) set public E set(int index, E element) Replaces the element at the specified position in this list with the specified element. Specified by: set in interface List¡E¿ Overrides: set in class AbstractSequentialList¡E¿ Parameters: index - index of the element to replace element - element to be stored at the specified position Returns: the element previously at the specified position Throws: IndexOutOfBoundsException - if the index is out of range (index ¡ 0 —— index ¿= size()) add public void add(int index, E element) Inserts the specified element at the specified position in this list. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices). Specified by: add in interface List¡E¿ Overrides: add in class AbstractSequentialList¡E¿ Parameters: index - index at which the specified element is to be inserted element - element to be inserted Throws: IndexOutOfBoundsException - if the index is out of range (index ¡ 0 —— index ¿ size()) remove public E remove(int index) Removes the element at the specified position in this list. Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the list. Specified by: remove in interface List¡E¿ Overrides: remove in class AbstractSequentialList¡E¿ Parameters: index - the index of the element to be removed Returns: the element previously at the specified position Throws: IndexOutOfBoundsException - if the index is out of range (index ¡ 0 —— index ¿= size()) indexOf public int indexOf(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. More formally, returns the lowest index i such that (o==null ? get(i)==null : o.equals(get(i))), or -1 if there is no such index. Specified by: indexOf in interface List¡E¿ Overrides: indexOf in class AbstractList¡E¿ Parameters: o - element to search for Returns: the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element lastIndexOf public int lastIndexOf(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element. More formally, returns the highest index i such that (o==null ? get(i)==null : o.equals(get(i))), or -1 if there is no such index. Specified by:

lastIndexOf in interface List¡E¿ Overrides: lastIndexOf in class AbstractList¡E¿ Parameters: o - element to search for Returns: the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element peek public E peek() Retrieves, but does not remove, the head (first element) of this list. Specified by: peek in interface Deque¡E¿ Specified by: peek in interface Queue¡E¿ Returns: the head of this list, or null if this list is empty Since: 1.5 element public E element() Retrieves, but does not remove, the head (first element) of this list. Specified by: element in interface Deque¡E¿ Specified by: element in interface Queue¡E¿ Returns: the head of this list Throws: NoSuchElementException - if this list is empty Since: 1.5 poll public E poll() Retrieves and removes the head (first element) of this list. Specified by: poll in interface Deque¡E¿ Specified by: poll in interface Queue¡E¿ Returns: the head of this list, or null if this list is empty Since: 1.5 remove public E remove() Retrieves and removes the head (first element) of this list. Specified by: remove in interface Deque¡E¿ Specified by: remove in interface Queue¡E¿ Returns: the head of this list Throws: NoSuchElementException - if this list is empty Since: 1.5 offer public boolean offer(E e) Adds the specified element as the tail (last element) of this list. Specified by: offer in interface Deque¡E¿ Specified by: offer in interface Queue¡E¿ Parameters: e - the element to add Returns: true (as specified by Queue.offer(E)) Since: 1.5 offerFirst public boolean offerFirst(E e) Inserts the specified element at the front of this list. Specified by: offerFirst in interface Deque¡E¿ Parameters: e - the element to insert Returns: true (as specified by Deque.offerFirst(E)) Since: 1.6 offerLast public boolean offerLast(E e) Inserts the specified element at the end of this list. Specified by: offerLast in interface Deque¡E¿ Parameters: e - the element to insert Returns: true (as specified by Deque.offerLast(E)) Since: 1.6 peekFirst public E peekFirst() Retrieves, but does not remove, the first element of this list, or returns null if this list is empty. Specified by: peekFirst in interface Deque¡E¿ Returns: the first element of this list, or null if this list is empty Since: 1.6 peekLast public E peekLast() Retrieves, but does not remove, the last element of this list, or returns null if this list is empty. Specified by: peekLast in interface Deque¡E¿ Returns: the last element of this list, or null if this list is empty Since: 1.6 pollFirst public E pollFirst() Retrieves and removes the first element of this list, or returns null if this list is empty. Specified by: pollFirst in interface Deque¡E¿ Returns: the first element of this list, or null if this list is empty Since: 1.6 pollLast public E pollLast() Retrieves and removes the last element of this list, or returns null if this list is empty. Specified by: pollLast in interface Deque¡E¿ Returns: the last element of this list, or null if this list is empty Since: 1.6 push public void push(E e) Pushes an element onto the stack represented by this list. In other words, inserts the element at the front of this list. This method is equivalent to addFirst(E).

Specified by: push in interface Deque¡E¿ Parameters: e - the element to push Since: 1.6 pop public E pop() Pops an element from the stack represented by this list. In other words, removes and returns the first element of this list. This method is equivalent to removeFirst().

Specified by: pop in interface Deque¡E¿ Returns: the element at the front

of this list (which is the top of the stack represented by this list) Throws: NoSuchElementException - if this list is empty Since: 1.6 removeFirstOccurrence public boolean removeFirstOccurrence(Object o) Removes the first occurrence of the specified element in this list (when traversing the list from head to tail). If the list does not contain the element, it is unchanged. Specified by: removeFirstOccurrence in interface Deque¡E¿ Parameters: o - element to be removed from this list, if present Returns: true if the list contained the specified element Since: 1.6 removeLastOccurrence public boolean removeLastOccurrence(Object o) Removes the last occurrence of the specified element in this list (when traversing the list from head to tail). If the list does not contain the element, it is unchanged. Specified by: removeLastOccurrence in interface Deque¡E¿ Parameters: o - element to be removed from this list, if present Returns: true if the list contained the specified element Since: 1.6 listIterator public ListIterator¡E¿ listIterator(int index) Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list. Obeys the general contract of List.listIterator(int). The list-iterator is fail-fast: if the list is structurally modified at any time after the Iterator is created, in any way except through the list-iterator's own remove or add methods, the list-iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Specified by: listIterator in interface List¡E¿ Specified by: listIterator in class AbstractSequentialList¡E¿ Parameters: index - index of the first element to be returned from the list-iterator (by a call to next) Returns: a ListIterator of the elements in this list (in proper sequence), starting at the specified position in the list Throws: IndexOutOfBoundsException - if the index is out of range (index ¡ 0 —— index ¿ size()) See Also: List.listIterator(int) descendingIterator public Iterator¡E¿ descendingIterator() Description copied from interface: Deque Returns an iterator over the elements in this deque in reverse sequential order. The elements will be returned in order from last (tail) to first (head). Specified by: descendingIterator in interface Deque¡E¿ Returns: an iterator over the elements in this deque in reverse sequence Since: 1.6 clone public Object clone() Returns a shallow copy of this LinkedList. (The elements themselves are not cloned.) Overrides: clone in class Object Returns: a shallow copy of this LinkedList instance See Also: Cloneable toArray public Object[] toArray() Returns an array containing all of the elements in this list in proper sequence (from first to last element). The returned array will be "safe" in that no references to it are maintained by this list. (In other words, this method must allocate a new array). The caller is thus free to modify the returned array.

This method acts as bridge between array-based and collection-based APIs.

Specified by: toArray in interface Collection¡E¿ Specified by: toArray in interface List¡E¿ Overrides: toArray in class AbstractCollection¡E¿ Returns: an array containing all of the elements in this list in proper sequence See Also: Arrays.asList(Object[]) toArray public ¡T¿ T[] toArray(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array. If

the list fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this list. If the list fits in the specified array with room to spare (i.e., the array has more elements than the list), the element in the array immediately following the end of the list is set to null. (This is useful in determining the length of the list only if the caller knows that the list does not contain any null elements.)

Like the toArray() method, this method acts as bridge between array-based and collection-based APIs. Further, this method allows precise control over the runtime type of the output array, and may, under certain circumstances, be used to save allocation costs.

Suppose x is a list known to contain only strings. The following code can be used to dump the list into a newly allocated array of String:

String[] y = x.toArray(new String[0]); Note that toArray(new Object[0]) is identical in function to toArray(). Specified by: toArray in interface Collection¡E¿ Specified by: toArray in interface List¡E¿ Overrides: toArray in class AbstractCollection¡E¿ Parameters: a - the array into which the elements of the list are to be stored, if it is big enough; otherwise, a new array of the same runtime type is allocated for this purpose. Returns: an array containing the elements of the list Throws: ArrayStoreException - if the runtime type of the specified array is not a supertype of the runtime type of every element in this list NullPointerException - if the specified array is null

5.8 Zirkulare LinkedList

5.9 Zweifach verknüpfte LinkedList

5.10 Dictionary

5.11 Hashing

5.12 HashMap

5.13 MapTree

5.14 LinkedMap

5.15 Sets

5.16 Binäre Bäume

5.17 Graphen

5.18 AVL Tree

Kapitel 6

Algorithmen

Sortier Algorithmen und Such Algorithmen

6.1 Breitensuche

6.2 Tiefensuche

6.3 Bubble Sort

6.4 Selection Sort

6.5 Shellsort

6.6 Mergesort

6.7 Quicksort

6.8 Sequential Suche

6.9 Binäres Suchen

6.10 Suchen nach Minimum und Maximum

6.11 Rucksackproblem

6.12 Greedy Algorithm

Kapitel 7

Aufgaben

7.1 Binäres Suchen

> **Aufgabe 1.** *Gegeben ist ein Array mit ganzen Zahlen. Gib den Index des gegebenen Keys. Falls kein Ergebnis gefunden wurde, gib -1 zurück.*
>
> *Beispiel: Gegeben ist folgender Array, wenn der Key 47 ist, dann soll die Binäre Suche 2 zurückgeben.*

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Key	*23*	*31*	*47*	*65*	*69*	*73*	*75*	*89*	*91*	*93*

Stichwörter: Array, Binary Search, Suche, Sliding Window, Zeiger, Pointer, Divide and Conquer, Teile und Herrsche

Vorgehensweise

1. Betrachte das Array von Anfang bis Ende
2. Berechne den Index der in der Mitte liegt
3. Wenn der Mitte Index genau auf den Key zeigt, dann gib den diesen Index zurück
4. Wenn das Element an der Mitte Index kleiner ist als der Key, dann betrachte nur das Subarray von der Mitte Index bis zum Ende
5. Wenn das Element an der Mitte Index größer ist als der Key, dann betrachte nur das Subarray von der Mitte Index bis zum Anfang
6. Wiederhole die obigen Schritte solange bis das Subarray leer ist

```
1 //a is sorted array
2 let binarySearch = function(a, key) {
3   let newA = a;
```

```
4   let currentIndex;
5   while(newA.length !== 0) {
6     currentIndex = newA.length / 2;
7     const currentKey = newA[currentIndex];
8
9     if (currentKey === key) {
10      return currentIndex;
11    } else if (currentKey < key) {
12      newA = newA.(currentIndex);
13    } else {
14      newA = newA.slice(0, currentIndex + 1);
15    }
16  }
17  return -1;
18 };
```

Listing 7.1: My Javascript Example

- Ein Array ist index basiert beginned bei 0

  Das weißt wahrscheinlich jeder bisher. Jedoch wird die Implikationen dessen sehr oft vergessen. Um auf den Index `currentIndex` zu kommen muss die `(arr.length - 1)/ 2` genommen werden.

- Array Index sind immer ganze Zahlen

  Auch dies sollte mittlerweile bekannt sein. Doch wie im obigen Fall zu sehen wurde durch `2` geteilt. Bei ungeraden Zahlen entstehen dadurch rationale Zahlen, die auf ganze Zahlen wieder zurück gecastet werden müssen

- Es wird nach dem Index des original Arrays gefragt

  Ein neues Array zu erstellen und darauf den Index zu ermitteln bedeutet, dass man den Index des neuen Arrays zurück gibt. Gesucht ist aber der Index des gegebenen Arrays

- `currentIndex` nicht wiederholt überprüfen

  Selbst wenn es möglich sein sollte das original Array in Stücke so zu teilen dass dennoch die original Index beizubehalten, ist die Auswahl der Indexe falsch, denn `currentIndex` wurde bereits geprüft. Es ist daher unsinnig, dass dieser Index noch im Subarray erscheint. Man müsste also `currentIndex + 1` für `currentKey < key` bzw. `curentIndex` (`currentIndex` selbst wird also nicht betrachtet) für `currentKey > key` wählen.

- Programm liefert kein Ergebnis

  Das Programm läuft in Endlosschleife und kann somit kein Ergebnis liefern.

Lösung Wir versuchen immer am Anfang einen "early exit" zu erreichen, damit der Code effizient bleibt. Alternativ hätte man auch `includes()` verwenden können. Aber `includes()` geht durch alle Array Elemente durch und hätte damit eine Laufzeit von `O(n)`.

Wir erreichen einen "early exit", indem wir überprüfen, ob das Array keine Elemente enthält oder ein Element. Bei keinem Element geben wir sofort `-1` zurück. Wenn ein Element vorhanden ist können wir gleich überprüfen, ob dieses Element dem `key` entspricht. Falls ja, geben wir `0` zurück, falls nicht, ist es nicht vorhanden und wir geben `-1` zurück.

Die Vorgehensweise ist ähnlich wie oben beschrieben. Wir werden aber zwei Zeiger benutzen. Dadurch bleiben die Indexe immer die der originalen Arrays.

Die Laufzeit

Zeit komplexität

```
1  //a is sorted array
2  let binarySearch = function(a, key) {
3    if (a.length === 0) {
4      return -1;
5    }
6    if (a.length === 1) {
7      return a[0] === key ? 0 : -1;
8    }
9    let startIndex = 0;
10   let endIndex = a.length - 1;
11   while(startIndex <= endIndex) {
12     const currentIndex = (startIndex + endIndex) / 2 | 0;
13     const currentKey = a[currentIndex];
14     if (currentKey === key) {
15       return currentIndex;
16     } else if (currentKey < key) {
17       startIndex = currentIndex + 1;
18     } else {
19       endIndex = currentIndex - 1;
20     }
21   }
22   return -1;
23 };
```

Listing 7.2: My Javascript Example

Aufgabe 2. *Gegeben ist ein Array mit sortierten Zahlen, die keine Duplikate enthält. Dieses Array ähnelt dem Array in der Aufgabe darüber. Der Unterschied hier ist, dass dieses Array um eine zufällige Stellenanzahl verschoben wurde. Es soll eine Zahl darin gefunden und dessen Index zurückgegeben werden. Falls sie nicht existiert, dann soll `-1` zurück gegeben werden. Die Lösung muss schneller als lineare Laufzeit sein.*

Beispiel: Wir nehmen das Beispiel aus der vorherigen Aufgabe.

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Key	*23*	*31*	*47*	*65*	*69*	*73*	*75*	*89*	*91*	*93*

Dieses Array verschieben wir um drei Stellen nach rechts:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Key	*89*	*91*	*93*	*23*	*31*	*47*	*65*	*69*	*73*	*75*

Hier ist die gesuchte Antwort 5, weil 47 am Index 5 ist.

Stichwörter: Array, Binary Search, Suche, Sliding Window, Zeiger, Pointer, Divide and Conquer, Teile und Herrsche

Vorgehensweise Ansätze um zuerst die Sollbruchstelle zu finden sind zu aufwändig.

1. Wir betrachten die Fälle, dass das Array
   (a) bis zum Anfang nicht gefunden wurde
   (b) bis zum Ende nicht gefunden wurde
   (c) außerhalb der Kontinuität liegt
2. Ermittelt den Wert in der Mitte des Arrays
3. Teile ihn dann in zwei Arrays auf
4. Suche in jeweils der zwei Subarrays nach der Zahl in Binary Search

```
let binarySearchSwitched = function(arr, key) {
  const maxIndex = arr.length - 1;
  let startPointer = 0;
  let endPointer = maxIndex;
  while (startPointer <= endPointer) {
    const currentPointer = (startPointer + endPointer) / 2 | 0;
    const oldCurrentKey = arr[currentPointer];

    if (oldCurrentKey < key) {
      // go to right
      startPointer = currentPointer + 1;
      const newCurrentPointer = (startPointer + endPointer) / 2;
      const newCurrentKey = arr[newCurrentPointer];
      if (newCurrentKey < oldCurrentKey) {
        // we are on the right; newCurrentKey has to be bigger than
            oldCurrentKey
        // if it is not, then we have to look at the other side
        return searchBinay(0, currentPointer - 1);
      }
    } else if (oldCurrentKey > key) {
      // go to left
      endPointer = currentPointer - 1;
      const newCurrentPointer = (startPointer + endPointer) / 2;
      const newCurrentKey = arr[newCurrentPointer];
      if (newCurrentKey > oldCurrentKey) {
        // we are on the left; newCurrentKey has to be smaller than
            oldCurrentKey
        // if it is not, then we have to look at the other side
        return searchBinay(currentPointer + 1, maxIndex);
      }
    } else {
      return currentPointer;
```

```
31      }
32
33      // search reaches end
34      if (startPointer > maxIndex || endPointer > maxIndex) {
35        return searchBinary(0, maxIndex / 2 | 0);
36      }
37      // search reaches start
38      if (startPointer < 0 || endPointer < 0 ) {
39        return searchBinary(maxIndex / 2 | 0, maxIndex);
40      }
41    }
42
43    function searchBinary(start, end) {
44      while (start <= end) {
45        const currentPointer = (start + end) / 2 | 0;
46        const oldCurrentKey = arr[currentPointer];
47        if (oldCurrentKey < key) {
48          // go to right
49          start = currentPointer + 1;
50        } else if (oldCurrentKey > key) {
51          // go to left
52          end = currentPointer - 1;
53        } else {
54          return currentPointer;
55        }
56      }
57      return -1;
58    }
59  };
```

Listing 7.3: My Javascript Example

Typische Fehler

- Zu lang, zu kompliziert und zu spezifisch

  Die Länge des Codes ist geschuldet durch die Tatsache, dass im Code spezifisch auf viele mögliche Fälle (Es wird das bzw. den Ende/Anfang erreicht ohne das Element gefunden zu haben; das neue Element ist außerhalb der Kontinuität der Reihe) eingegangn wird und versucht wird diese durch viele `if` Abfragen abzufangen. Normalerweise ist das schon ein erster Indiz, dafür dass diese Lösung nicht optimal ist.

- Nutzt vorhandene Informationen nicht

  Offensichtlich ist es wichtig zu wissen, wo der Schnitt sich befindet bzw. welche Elemente vor dem Schnitt sich befinden. Man hätte ausnutzen können, dass die Zahlen sortiert sind und dadurch die Elemente, die vor dem Schnitt sind damit erkennen können. Dies wurde im obigen Ansatz aber nicht berücksichtigt. Deshalb liefer er für folgendes Array und folgenden Key einen Fehler:

  `[17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]` mit der gesuchten Zahl `20`

Obiger Ansatz gibt inkorrekterweise `-1` zurück, obwohl `3` erwartet worden wäre.

Lösung Auf die Lösung zu kommen ist nicht sofort ersichtlich. Aber die Lösung zu dieser Aufgabe ist an sich recht einfach, wenn man sich folgende Dinge vergegenwärtigt:

- Teilt man das Array in der Mitte, so ist mindestens eines der Hälfte immer sortiert.
- Weil das Array sortiert ist, können wir vergleichen vergleichen, ob die Zahl in einen bestimmten Abschnitt liegt oder nicht, indem wir einen Arrayabschnitt nehmen und das erste und das letzte Element aus dem Teilarray mit der Zahl vergleichen.

Die Laufzeit
Zeit komplexität

```
1  function keyInFirstHalfSortedArray(start, mid, key, arr) {
2    return arr[start] < arr[mid] && arr[start] >= key && arr[mid] <
         key;
3  }
4  function keyInSecondHalfSortedArray(mid, end, key, arr) {
5    return arr[mid] < arr[end] && arr[mid] < key && arr[end] >= key;
6  }
7  function keyInFirstHalfUnsortedArray (start, mid, key, arr) {
8    return arr[start] > arr[mid] && arr[start] >= key || arr[mid] <
         key;
9  }
10 function keyInFirstHalfUnsortedArray(mid, end, key, arr) {
11   return arr[mid] > arr[end] && arr[mid] > key || arr[end] <= key;
12 }
13 let binarySearchSwitched = function(arr, key) {
14   const maxIndex = arr.length - 1;
15   let startPointer = 0;
16   let endPointer = maxIndex;
17
18   while (startPointer <= endPointer) {
19     const midPointer = (startPointer + endPointer) / 2 | 0;
20     if (arr[midPointer] === key) {
21       return midPointer;
22     }
23     if (keyInFirstHalfSortedArray(startPointer, midPointer, key,
           arr)) {
24       endPointer = midPointer - 1;
25     } else if (keyInSecondHalfSortedArray(midPointer, endPointer,
           key, arr)) {
26       startPointer = midPointer + 1;
27     } else if (keyInFirstHalfUnsortedArray(startPointer, midPointer
           , key, arr)) {
28       endPointer = midPointer - 1;
29     } else if (keyInFirstHalfUnsortedArray(midPointer, endPointer,
           key, arr)) {
30       startPointer = midPointer + 1;
31     } else {
```

```
32        return -1;
33      }
34    }
35    return -1;
36  };
```

Listing 7.4: My Javascript Example

Eine alternative Lösung wäre es Rekursion zu benutzen:

```
1  let binarySearchRecursive = function(arr, st, end, key) {
2    // assuming all the keys are unique.
3    if (st > end) {
4      return -1;
5    }
6
7    let mid = st + Math.floor((end - st) / 2);
8
9    if (arr[mid] === key) {
10     return mid;
11   }
12
13   if (arr[st] <= arr[mid] && key <= arr[mid] && key >= arr[st]) {
14     return binary_search_recs(arr, st, mid - 1, key);
15   } else if (arr[mid] <= arr[end] && key >= arr[mid] && key <= arr[
       end]) {
16     return binary_search_recs(arr, mid + 1, end, key);
17   } else if (arr[end] <= arr[mid]) {
18     return binary_search_recs(arr, mid + 1, end, key);
19   } else if (arr[st] >= arr[mid]) {
20     return binary_search_recs(arr, st, mid - 1, key);
21   }
22
23   return -1;
24 };
25
26 let binarySearchSwitched = function(arr, key) {
27   return binarySearchRecursive(arr, 0, arr.length - 1, key);
28 };
```

Listing 7.5: My Javascript Example

JS ist jedoch schlecht für Rekursion ausgelegt, wie ich es in einen der vorherigen Kapitel beschrieben habe.

Aufgabe 3. *Gegeben ist ein sortiertes Array von ganzen Zahlen. Gib den niedrigsten und höchsten Index einer Zahl zurück mit jeweils einer eigenen Funktion. Falls die Zahl nicht existiert, dann gib -1 zurück. Das Array kann beliebig lang sein und eine beliebige Anzahl an Duplikaten enthalten.*

Beispiel: Gegeben ist folgendes Array[a] und eine Zahl 3.

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Value	*1*	*1*	*3*	*3*	*3*	*3*	*3*	*3*	*11*	*13*	*17*

Der niedrigste Index ist 2 und der höchste Index ist 7

[a] Die farbliche Markierung ist nur zur Verdeutlichung wo der niedrigste und höchste Index anfängt bzw. aufhört.

Stichwörter: Array, Zeiger, Pointer, Suchen, Binary search

Vorgehensweise Zur Lösung dieser Aufgabe werden zwei Funktionen verwendet.

Für die Funktion zur Berechnung des niedrigsten Index:

1. Wir iterieren mit einer Schleife, bis wir das erste Mal auf die Zahl im Array treffen und geben den Index dann zurück

2. Falls es keine Übereinstimmung gibt, dann geben wir `-1` zurück

Für die Funktion zur Berechnung des höchsten Index:

1. Wir ermitteln den niedrigsten Index anhand der Funktion oben. Falls keins existiert geben wir `-1` zurück, denn wenn es keinen niedrigsten Index existiert, existiert auch kein höchsten Index.

2. Wir verwenden auch hier eine Schleife. Die Schleife startet beim niedrigsten Index, die wir oben ermittelt haben.

3. Wir iterieren solange bis wir entweder das Arrayende erreicht haben oder bis wir auf eine Zahl treffen, die ungleich der gegebenen Zahl ist. Der Rückgabewert ist der aktuelle Index dekrementiert um `1`, weil der Index um `1` höher ist als der gesuchte Index

```
let find_low_index = function(arr, key) {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === key) {
      return i;
    }
  }
  return -1;
};

let find_high_index = function(arr, key) {
  let lowIndex = find_low_index(arr, key);
  if (lowIndex === -1) {
    return -1;
  }
  let highIndex = lowIndex;
  for (highIndex = lowIndex; highIndex < arr.length; highIndex++) {
    if (arr[highIndex] !== key) {
      return --highIndex;
    }
  }
  return --highIndex;
};
```

Listing 7.6: My Javascript Example

Typische Fehler

- Schlechte Laufzeit

  Wie oben erwähnt, kann der Array sehr lang sein. Die Laufzeit ist hier linear. Weil das Array sortiert ist, wäre ein Binary Search effizienter

Bei großen Datenmengen, die sortiert sind, ist es der Binary Search Ansatz sinnvoll zu verwenden.

Zur Ermittlung des niedrigsten Index:

1. Wir nehmen ein Element von der Mitte des Arrays und vergleichen es mit der gegebenen Zahl. Ist es kleiner, betrachten wir den rechten Arrayteil. Ist es größer oder gleich der gegebenen Zahl, betrachten wir den linken Arrayteil.

2. Vom betrachteten Arrayteil nehmen wir wieder ein Element von der Mitte und vergleichen es mit der gegbenen Zahl und verfahren wie im ersten Schritt.

3. Diese Schritte wiederholen wir solange, bis am Ende wir nur ein Element haben, dass der Zahl entspricht oder wir gar kein Element mehr finden. Im ersten Fall geben wir den Index zurück. Im zweiten Fall geben wir `-1` zurück.

Zur Ermittlung des höchsten Index:

1. Wir nehmen ein Element von der Mitte des Arrays und vergleichen es mit der gegebenen Zahl. Ist es kleiner, betrachten wir den rechten Arrayteil. Ist es größer oder gleich der gegebenen Zahl, betrachten wir den linken Arrayteil.

2. Vom betrachteten Arrayteil nehmen wir wieder ein Element von der Mitte und vergleichen es mit der gegbenen Zahl und verfahren

3. Diese Schritte wiederholen wir solange, bis am Ende wir nur ein Element haben, dass der Zahl entspricht oder wir gar kein Element mehr finden. Im ersten Fall geben wir den Index zurück. Im zweiten Fall geben wir `-1` zurück.

```
1  let find_low_index = function(arr, key) {
2    if (arr.length === 0) {
3      return -1;
4    }
5    let startIndex = 0;
6    let endIndex = arr.length - 1;
7
8    while (startIndex <= endIndex) {
9      const midIndex = (startIndex + endIndex) / 2 | 0;
10     const currentValue = arr[midIndex];
11     if (currentValue < key) {
12       startIndex = midIndex;
```

```
13      } else {
14        endIndex = midIndex;
15      }
16    }
17    if (arr[startIndex] === key) {
18      return startIndex;
19    }
20    return -1;
21  };
22
23  let find_high_index = function(arr, key) {
24    let lowIndex = find_low_index(arr, key);
25    if (lowIndex === -1) {
26      return -1;
27    }
28
29    let startIndex = lowIndex;
30    let endIndex = arr.length - 1;
31
32    while (startIndex <= endIndex) {
33      const midIndex = (startIndex + endIndex) / 2 | 0;
34      const currentValue = arr[midIndex];
35      if (currentValue === key) {
36        startIndex = midIndex;
37      } else {
38        endIndex = midIndex;
39      }
40    }
41    if (arr[endIndex] === key) {
42      return endIndex;
43    }
44    return endIndex - 1;
45  };
```

Listing 7.7: My Javascript Example

Typische Fehler

- Early exit unnötig

  In vorherigen Code Beispielen habe ich oft davon geredet einen early exit zu verwenden - also Code, das das Programm so schnell wie möglich beendet. In der Zeile 2 wurde so ein Pattern implementiert, das aber bei genauer Betrachtung unnötig ist:

  In der Zeile 8 sehen wir, dass die Schleife nur dann ausgeführt wird, wenn `startIndex <= endIndex` ist. Wäre die Länge des Arrays 0, dann wäre `startIndex` 0 und `endIndex` wäre -1. Damit ist die Bedingung in der `while` Schleife nicht erfüllt und wird am Ende -1 zurückgeben.

- Einige Werte werden mehr als nötig überprüft.

  In den Zeilen, wo `startIndex` und `endIndex` mit `midIndex` zugewiesen wird, wird damit auch `midIndex` nochmals geprüft. Dies ist ineffizient. Stattdessen weist man `startIndex` bzw. `endIndex` das Element danach oder davor.

Lösung Mit Berücksichtigung der oben genannten Punkte, schaut unser Lösungsvorschlag wie folgt aus. Die Vorgehensweise ist identisch zu oben.

```
1
2  let find_low_index = function(arr, key) {
3    let low = 0;
4    let high = arr.length - 1;
5    let mid = Math.floor(high / 2);
6
7    while (low <= high) {
8
9      let mid_elem = arr[mid];
10
11     if (mid_elem < key) {
12       low = mid + 1;
13     } else {
14       high = mid - 1;
15     }
16
17     mid = low + Math.floor((high - low) / 2);
18   }
19
20   if (arr[low] === key) {
21     return low;
22   }
23
24   return -1;
25 };
26
27 let find_high_index = function(arr, key) {
28   let low = 0;
29   let high = arr.length - 1;
30   let mid = Math.floor(high / 2);
31
32   while (low <= high) {
33     let mid_elem = arr[mid];
34
35     if (mid_elem <= key) {
36       low = mid + 1;
37     } else {
38       high = mid - 1;
39     }
40
41     mid = low + Math.floor((high - low) / 2);
42   }
43
44   if (arr[high] === key) {
45     return high;
46   }
47
48   return -1;
49 };
```

Listing 7.8: My Javascript Example

7.2 Maximum im gleitenden Fenster

Aufgabe 4. *Gegeben ist ein langes Array mit ganzen Zahlen. Zudem ist noch ein Fenster mit einer Breite* `w` *gegeben, das sich von Anfang bis Ende des Arrays bewegt. Finde alle Maxima, die im gleitenden Fenster auftauchen.*

Beispiel: Gegeben ist folgender Array mit einer Fensterbreite von `4`*.*

3	*23*	*-31*	*47*	*-65*	*69*

Das Fenster bewegt sich nun von links nach rechts und enthält folgende Zahlen:

1. *Schritt*

3	*23*	*-31*	*47*

   Das Maximum ist `47`

2. *Schritt*

23	*-31*	*47*	*-65*

   Das Maximum ist wieder `47`

3. *Schritt*

-31	*47*	*-65*	*69*

   Das Maximum ist wieder `69`

Das erwartete Ergebnis ist in diesem Beispiel

47	*47*	*69*

Stichwörter: Array, Suche, Sliding Window, Zeiger, Pointer, Dequeue

Vorgehensweise

1. Wir erstellen zwei Zeiger

   Die Zeiger nennen wir startWindow und endWindow. startWindow wird mit `0` und `endWindow` wird mit `array.length - 1` initialisiert.

2. Hilfsfunktion zur Berechnung des Maximums

   Wir erstellen noch eine Hilfsfunktion, die das Maximum für gegebenes `startWindow` und `endWindow` berechnet und in ein Resultat-Array hinzufügt.

3. Iteration bis Arrayende

   Wir iterieren dann duch die verbleibenden Elemente und inkrementieren `startWindow` und `endWindow`. Dabei rufen wir jedesmal die Hilfsfunktion zur Berechnung des Maximums.

4. Rückgabe

   Am Ende geben wir das Resultat-Array zurück

```
const findMaxSlidingSindow = function(arr, window_size) {
  const getMax = (startWindow, endWindow) => {
    const max = arr
    .slice(startWindow, endWindow + 1)
    .reduce((max, item) => item > max ? item : max, -Infinity);
    result.push(max);
  }
  const result = [];
  let startWindow = 0;
  let endWindow = window_size - 1;
  while (endWindow < arr.length) {

    getMax(startWindow++, endWindow++);
  }
  return result;
};
```

Listing 7.9: My Javascript Example

Typische Fehler

- Kein early exit verwendet

  Man sollte immer versuchen einen early exit zu verwenden, d.h. schauen, ob anhand einer einfachen überprüfung am Anfang gleich auf das richtige Ergebnis geschlossen werden kann. Beispw. muss man nichts berechnen, wenn das Array leer ist oder die gegebene Fensterbreite größer ist als das Array.

- Keine geeignete Datenstruktur verwendet

  Die Aufgabe bietet es sich an ein `Dequeue` zu verwenden. Dadurch brauchen wir auch keine zwei Zeiger.

- Ungünstige Laufzeit

  Durch die Hilfsfunktion zur Berechnung des Maximums werden bereits besuchte Elemente erneut besucht. Bei einer kleinen Fensterbreite ist das kein großes Problem aber angenommen es handelt sich um ein langes Array mit einer sehr großen Fensterbreite. Dies führt zu einer langsamen Laufzeit. Das ist ein erster Hinweis, dass der vorliegende Code nicht optimal ist.

Lösung Die Schwierigkeit liegt darin, das Maximum zu finden ohne erneut die Zahlen im Fenster durchzugehen. Idealerweise wollen wir nur eimal durch die Zahlen im Array gehen und dabei das Maximum speichern. Nachfolgend werde ich begründen, warum eine `Dequeue` die geeignete Datenstruktur ist.

Angenommen wir haben folgendes Array und eine Fensterbreite von `3` und benutzen dazu wir zunächst eine Variable zur Speicherung des Maximums:

7	6	2	4	1	7

Wenn wir durch die einzelnen Arrays gehen und das Maximum jedesmal in der Variable speichern, würden wir folgendes Ergebnis bekommen:

1. Schritt

7

   Erste Zahl ist 7. Dadurch ist max auch 7

2. Schritt

7	6

   Die 6 ist kleiner als das aktuelle max und daher ist max immer noch 7

3. Schritt

7	6	2

   Max bleibt nach der 2 immer noch bei 7. Dadurch haben wir die Fensterbreite erreicht. Das max ist für die erste Fensterbreite ist 7.

4. Schritt

6	2	4

   Ab jetzt bekommen wir eine konstante Anzahl an 3 Zahlen. Die 7 ist raus. Die 4 kommt rein. Das Maximum kann nun nicht mehr 7 sein. Aber was ist es stattdessen? Es kann nicht 4 sein. Es ist 6. Aber unser Ansatz hat keine Möglichkeit zu erkennen, dass es die 6 ist.

Was wir daher brauchen ist also keine Variable zur Speicherung des Maximums, sondern wir müssen eine Sequenz speichern. Wie wollen die Sequenz so speichern, dass die größere Zahl immer vorne steht. Die Zahlen im `Dequeue` werden von links nach rechts immer kleiner - die Zahlen links sind also immer größer als die Zahlen rechts. Wenn wir also das Maximum haben wollen, dann greifen wir immer auf die erste Zahl im `Dequeue`. Falls die Zahl außerhalb des Arrays rausgeht, dann löschen wir diese Zahl von der Spitze. Damit wir leichter wissen können, welche Zahl aus dem Fenster rausgeht, speichern wir daher nicht den Wert der Zahl, sondern den Index der Zahl.

Wir benutzen nachfolgend dieselbe Zahlenfolge. Diesmal habe ich noch die Indexe hinzugefügt.

Index:	0	1	2	3	4	5
Value:	7	6	2	4	1	7

Wenn wir obiges Beispiel nehmen, dann würde der Ablauf wie folg aussehen:

1. Schritt

Index:	0	1	2	3	4	5
Value:	7	6	2	4	1	7

   Erste Zahl ist 7. Weil die `Dequeue` noch leer ist, pushen wir das erste Index sie in unsere `Dequeue` rein | 0 |

2. Schritt

Index:	0	1	2	3	4	5
Value:	7	6	2	4	1	7

Die 6 ist kleiner (man erinnert sich: Wir wollen in der `Dequeue` die Zahlen der kleiner nach speichern) als das aktuelle max und wir pushen sie ins `Dequeue`: | 0 | 1 |

3. Schritt

Index:	0	1	2	3	4	5
Value:	7	6	2	4	1	7

Die 2 ist kleiner als die 6. Dadurch pushen wir auch sie in unsere `Dequeue` | 0 | 1 | 2 |.

Damit haben wir die Fensterbreite erreicht. Jetzt müssen wir aufpassen, wann eine Zahl aus dem Fenster rausrutscht. Falls diese Zahl nicht mehr im Fenster ist, müssen wir auch dessen Index rauslöschen.

4. Schritt

Index:	0	1	2	3	4	5
Value:	7	6	2	4	1	7

Die 4 ist in das Fenster reingekommen. Die 4 ist aber größer als die 2. Daher löschen wir sie (bzw. dessen Index). Wir überprüfen, ob vor der Zahl 2 noch eine Zahl ist, die kleiner ist als die 4. Dies ist nicht der Fall. Wäre es aber der Fall, so müssten wir auch diesen Index löschen. Wir müssten nämlich so lange die Indexe löschen, bis der `Dequeue` die Struktur der Gestalt hat, dass sie absteigend ist, also vorne die großen Zahlen, hinten die kleinen. Daher löschen wir nur die Index der Zahl 2 aus der `Dequeue` und fügen den Index der Zahl 4 hinzu:

| 0 | 1 | 3 |.

Es passiert an dieser Stelle aber noch mehr: Die 7 (Index Stelle 0) ist aus dem Fenster gerutscht. Das ist deshalb bemerkenswert, weil die 7 bisher das Maximum war und damit an erster Stelle unserer `Dequeue` steht. Weil die Zahl 7 jetzt raus ist, löschen wir auch die den Index 0 aus unserem `Dequeu` | 1 | 3 |. Dadurch steht der Index 1 an erster Stelle unserer `Dequeue`. D.h. die 6 ist das Maximum an dieser Stelle.

Ich könnte das jetzt weiter fortführen, aber ich glaube das Prinzip sollte mittlerweile klar sein.

Unter der Berücksichtigung der "typischen Fehler" und den Ansatz mit der `Dequeue`, schaut die Implementierung wie folgt aus:

```
1  const findMaxSlidingWindow = function(arr, windowSize) {
2    // early exit
3    if(arr.length == 0 || windowSize > arr.length) {
4      return;
5    }
6    let result = [];
7    let dequeue = [];
8    const dequeueIsNotEmpty = () => dequeue.length > 0;
9    const currentElementLargerThanLastDequeueElement = currentEl =>
         currentEl >= arr[dequeue[dequeue.length - 1]]
10
11   // setup for first step
```

```
12    for (let i = 0; i < windowSize; i++) {
13      // remove all elements that are smaller than the current
            element
14      while (dequeueIsNotEmpty()
15      && currentElementLargerThanLastDequeueElement(arr[i])) {
16        dequeue.pop();
17      }
18
19      dequeue.push(i);
20    }
21
22    result.push(arr[dequeue[0]])
23
24    // remaining steps
25    for (let i = windowSize; i < arr.length; i++) {
26      // remove all elements that are smaller than the current
            element
27      while (dequeueIsNotEmpty()
28      && currentElementLargerThanLastDequeueElement(arr[i])) {
29        dequeue.pop();
30      }
31
32      // if number falls out from the window, we have to delete it
            from the dequeue
33      if (dequeueIsNotEmpty() && (dequeue[0] < i - (windowSize - 1)))
            {
34        dequeue.shift();
35      }
36
37      dequeue.push(i);
38      result.push(arr[dequeue[0]]);
39    }
40
41    return result;
42  };
```

Listing 7.10: My Javascript Example

Eine alternative Lösung ist es, einen `Heap()` zu verwenden.

7.3 Die kleinste gemeinsame Zahl in verschiedenen Arrays

Aufgabe 5. *Gegeben sind drei Arrays, die ganze Zahlen enthalten. Die Zahlen sind aufsteigend sortiert. Gesucht ist die kleinste Zahl, die in allen diesen Arrays vorkommt. Falls es keine gemeinsame Zahl gibt, dann soll -1 zurückgegeben werden.*

Beispiel: Gegeben sind folgende drei Arrays und es wird nach der kleinsten Zahl gesucht, die alle der gegebenen Arrays gemeinsam haben.

1	*7*	*11*	*15*	*26*

3	*4*	*5*	*6*	*7*	*14*	*16*	*69*

2	*5*	*6*	*7*

Die kleinste Zahl, die in allen der obigen Arrays vorkommt ist die 7.

Stichwörter: Array, Suche, Zeiger

Vorgehensweise

1. Wir erstellen drei Zeiger - eines für jedes Array. Wir fangen bei `0` an, da die Arrays aufsteigend sortiert sind.

2. Danach gehen wir in eine Schleife rein. Die Schleife soll solange ausgeführt werden, bis eines der Zeiger über das Ende ihres jeweiligen Arrays erreicht. Denn wenn der Zeiger über dem Ende des Arrays erreicht ohne, dass wir eine gemeinsame Zahl gefunden haben, dann können wir davon ausgehen, dass es keine kleinste gemeinsame Zahl unter allen Arrays existiert.

3. Wir greifen zuerst das Element, auf das der Zeiger des ersten Array momentan zeigt. In der ersten Iteration ist es das erste Element. Innerhalb der äußeren Schleife haben wir noch eine zweite Schleife, die den zweiten Array durchläuft und nach dem aktuellen Wert im ersten Array durchsucht.

4. Wird das Element gefunden wird ein Flag gesetzt und die innere Schleife unterbrochen. Der Zeiger des zweiten Arrays wird dann auf den letzten Wert der Laufvariable gesetzt, damit er bei der nächsten Iteration der äußeren Schleife wieder dort fortsetzen kann, wo er aufgehört hat.

5. Danach kommt eine zweite innere Schleife. Diese wird aber nur durchlaufen, wenn in der ersten inneren Schleife ein gemeinsamer Wert zwischen dem ersten und dem zweiten Array gefunden wurde, also wenn der Flag gesetzt wurde.

6. Falls der Flag gesetzt wurde, wird die zweite innere Schleife durchlaufen. Innerhalb dieser Schleife ist es analog zur ersten inneren Schleifen mit dem Unterschied, dass, wenn ein übereinstimmender Wert gefunden wurde, dann wird dieser Wert sofort zurückgegeben, denn damit wurde ein gemeinsamer Wert unter allen drei Arrays gefunden. Falls nicht, also falls ein Wert gefunden wurde, der größer ist als der aktuelle Vergleichswert, dann verläuft es genauso wie in der ersten inneren Schleife: Der Zeiger wird auf dem aktuellen Laufindex gesetzt und die Schleife wird anschließend unterbrochen.

7. Am Ende der äußeren Schleife wird der Zeiger des ersten Arrays um eins erhöht und die nächste Iteration der äußeren Schleife wird durchlaufen.

8. Falls die äußere Schleife komplett durchlaufen wird und kein übereinstimmendes Element gefunden wurde, dann wird `-1` zurückgegeben.

```
let findLeastCommonNumber = function(a, b, c) {
  let aPointer = 0;
  let bPointer = 0;
  let cPointer = 0;

  while (aPointer < a.length ||
       bPointer < b.length ||
       cPointer < c.length) {
    const aValue = a[aPointer];
    let bFound = false;
    for (let i = bPointer; i < b.length; i++) {
      if (b[i] === aValue) {
        bPointer = i;
        bFound = true;
        break;
      }
      if (b[i] > aValue) {
        bPointer = i;
        break;
      }
    }
    if (bFound) {
      for (let i = cPointer; i < c.length; i++) {
        if (c[i] === aValue) {
          return a[aPointer];
        }
        if (c[i] > aValue) {
          cPointer = i;
          break;
        }
      }
    }
    aPointer++;
  }
  return -1;
};
```

Listing 7.11: My Javascript Example

Typische Fehler

- Kein early exit verwendet

  Man hätte hier überprüfen können, ob mindestens eines der Arrays leer ist und gegebenfalls sofort `-1` zurückgeben können.

- Relativ komplex

  Durch die verschachtelte Schleife und die verschachtelten `if` Abfragen ist der Code etwas komplexer als notwendig und somit vergleichsweise schwerer verständlich als die nachfolgende Lösung. Generell sollte man versuchen die Verschachtelungen so gering wie möglich zu halten. Dies erhöht die Verständlichkeit und damit auch die Wartbarkeit.

Lösung Die obige Lösung liefert ebenfalls das richtige Ergebnis. Nachfolgend aber eine etwas leichter verständliche Lösung.

1. Wir können hier einen early exit anwenden. Denn wenn eines der Array leer ist, dann können wir schlussfolgern, dass es keinen gemeinsamen Wert unter allen drei Arrays gibt. Wir geben in diesem Fall also `-1`

2. Wir erstellen wieder drei Zeiger - eines für jedes Array. Wir fangen auch hier bei `0` an.

3. danach gehen wir in eine Schleife rein. Bis hier ist noch alles gleich. Im Gegensatz zum Erstversuch aber, überprüfen wir in der Schleife, ob alle Zeiger noch kleiner sind als die Länge der jeweiligen Arrays. Falls eines der Zeiger über das Ende des Arrays erreicht hat, dann terminiert die Schleife. Diese Änderung hat damit zu tun, was wir später im Schleifenrumpf machen werden.

4. Das erste, was wir im Schleifenrumpf überprüfen ist, ob die Werte, auf die alle Zeiger momentan zeigen alle gleich sind. Falls dem so ist, geben wir den Wert zurück.

5. Im nächsten Schritt überprüfen wir

6. Wir auch im dritten Array dieselbe Zahl gefunden, dann geben wir sie zurück. Ansonsten erhöhen wir den Zeiger des ersten Arrays und führen die Schritte ab `3` wieder aus.

7. Dies wiederholen wir bis wir

   Am Ende geben wir das Resultat-Array zurück

```
let find_least_common_number = function(a, b, c) {
  if (a.length === 0 ||
    b.length === 0 ||
    c.length === 0) {
    return -1;
  }
  let i = 0;
  let j = 0;
  let k = 0;

  while (i < a.length
  && j < b.length
  && k < c.length) {

    // Finding the smallest common number
    if (a[i] === b[j]
    && b[j] === c[k]) {
      return a[i];
    }

    // Let's increment the iterator
    // for the smallest value.
```

```
23
24      if (a[i] <= b[j]
25      && a[i] <= c[k]) {
26        i++;
27      } else if (b[j] <= a[i]
28      && b[j] <= c[k]) {
29        j++;
30      } else if (c[k] <= a[i]
31      && c[k] <= b[j]) {
32        k++;
33      }
34    }
35
36    return -1;
37  };
```

Listing 7.12: My Javascript Example

Aber auch diese Lösung ist nicht perfekt. Ein Nachteil ist, dass diese Lösung, im Gegensatz zum Erstversuch, weniger skalierbarer ist. Angenommen die Anforderungen ändern sich und

7.4 Array verschieben

Aufgabe 6. *Gegeben ist ein langes Array und eine ganze Zahl. Die Zahl steht dafür um wieviel das Array verschoben wird. Das Vorzeichen der Zahl steht für die Richtung in welcher das Array verschoben wird. Eine Zahl größer als 0 heißt verschieben nach rechts. Ein Zahl kleiner als 0 heißt verschieben nach links. Gesucht ist das Resultat nach dem Verschieben.*

Beispiel: Gegeben ist folgender Array und eine Zahl 4.

3	23	-31	47	-65	69

Nach dem Verschieben um 4 nach rechts ist das erwartete Ergebnis:

-31	47	-65	69	3	23

Nehmen wir dasselbe Array. Verschieben wir das Array um -1, dann schaut das erwartete Ergebnis folgendermaßen aus:

23	-31	47	-65	69	3

Stichwörter: Array, Zeiger, Pointer

Vorgehensweise

1. Wir teilen das Problem in drei Fälle:
   - wenn die Zahl gleich 0 oder gleich die Arraylänge ist
   - wenn die Zahl größer 0 ist
   - wenn die Zahl kleiner 0 ist
2. Wenn die Zahl gleich 0 oder gleich der Arraylänge ist, dann wissen wir, dass wir die Element nicht verschieben müssen. In diesem Fall können wir das Original Array ohne Änderung zurückgeben

3. Wenn die Zahl größer als 0 ist, dann will ich das Array nach rechts verschieben. Alle Elemente werden also um `n` Stellen nach rechts verschoben. Am Ende der Operation sollen die `n` letzten Elemente vorne stehen. Dazu kopiere ich mir die ersten `Array.length - n` Elemente und hänge sie hinten an. Danach lösche ich die ersten `Array.length - n` Elemente. Dadurch stehen die letzten `n` Elemente vorne am Array.

4. Wenn die Zahl kleiner als 0 ist, heißt das, dass ich die Elemente nach links verschiebe. Alle Elemente werden also um den Betrag von `n` Stellen nach links verschoben. Wir bilden also zuerst den Betrag von `n`. Dann hängen wir die ersten `n`[1] Elemente werden hinten angehängt. Dazu kopieren wir uns die ersten `n` Elemente und hängen sie hinten an. Danach löschen wir die ersten `n` Elemente wieder.

```
1  let rotate_array = function(arr, n) {
2    if (n === 0 || n === arr.length) {
3      return arr;
4    }
5    if (n > 0) {
6      const len = arr.length;
7      for (let i = 0; i < len - n; i++) {
8        arr.push(arr[i]);
9      }
10     arr.splice(0, len - n);
11     return arr;
12   }
13   if (n < 0) {
14     const absN = Math.abs(n);
15     for (let i = 0; i < absN; i++) {
16       arr.push(arr[i]);
17     }
18     arr.splice(0, absN);
19     return arr;
20   }
21 };
```

Listing 7.13: My Javascript Example

Typische Fehler

- Zu lang bzw. sehr viel Wiederholungen

  Der Code ist länger als nötig. Dies liegt daran, dass bestimmte Codestellen wiederholt werden, wie z.B. `return arr;` oder die beiden `for` Schleifen. Auch ist die Abfrage, ob `n === arr.length` überflüssig, denn für den Fall, dass `n` gleich der Arraylänge ist, dann kommt es in den Fall `(n > 0)` rein und in der `for` Schleife steht dann

  `(let i = 0; i < 0; i++).`

  Die Schleife wird also gar nicht erst ausgeführt. Weiter unten würde dann dieser Ausdruck stehen:

[1] Ab dieser Stelle kennzeichnet `n` den Betrag von `n`. `n` ist also immer positiv

`arr.splice(0, 0);`

Das original Array bleibt also unberührt.

Wenn man die `return arr;` aus den `if` Ausdrücken herauszieht und es ganz unten an der Funktion anhängt, kann man die ersten drei Zeilen löschen ohne dass die Ausgabe des Programms verändert wird:

```
1  if (n > 0) {
2    const len = arr.length;
3    for (let i = 0; i < len - n; i++) {
4      arr.push(arr[i]);
5    }
6    arr.splice(0, len - n);
7  }
8  if (n < 0) {
9    const absN = Math.abs(n);
10   for (let i = 0; i < absN; i++) {
11     arr.push(arr[i]);
12   }
13   arr.splice(0, absN);
14 }
15 return arr;
```

Listing 7.14: My Javascript Example

- Ineffizient

  Angenommen man will will das Array um `1` nach rechts schieben, dann muss man man `Array.length - 1` Elemente kopieren und diese hinten anhängen und dann dieselbe Anzahl vorne löschen. Wie man sich vorstellen kann ist das bei langen Arrays ineffizient.

- Wenig intuitiv

  Code wird öfter gelesen als dass sie geschrieben wird. Daher ist es wichtig, dass Code für andere Entwickler leicht verständlich ist. In diesem Fall ist die Berechnung `Array.length - n` nicht sofort verständlich.

Eine anderer Ansatz:

Wir wissen, dass, wenn `n` positiv ist, dann verschiebt sich das Array um `n` Stellen nach rechts. Das Element an der `0`ten Stelle befindet sich nach dem Verschieben an der `n`ten Stelle rechts. Die nun freien Stellen links von dem ursprünglich ersten Element, werden mit den Elementen aufgefüllt, die ganz rechts sind. Wir schneiden also vor dem `Array.length - n`-ten Element (`0` basierter Index) ab und hängen den rechten Teil an das Linke.

Wenn `n` negativ ist, dann verschiebt sich das Array um `n` Stellen nach links. Die dadurch freien Stellen rechts werden mit den Elementen ganz links aufgefüllt. Das Element an der ursprünglich `0`ten Stelle befindet sich nach dem Verschieben an der `Array.length + n`ten [2] Stelle (`0` basierter Index). Wir schneiden also vor dem `Array.length - (Array.length + n)`-ten Element ab und hängen den linken Teil an das Rechte.

[2] Wir addieren hier weil `n` negativ ist

Wir können das Verschieben des Arrays also erreichen, indem wir das Array an einer bestimmten Stelle "zerschneiden" und den zerschnittenen Teil entweder links oder rechts anhängen - abhängig davon, ob die Zahl größer oder kleiner `0` ist.

1. Wir nutzen die `splice()` Funktion, um das Original Array ab bestimmten Elementen zu zerteilen. `splice()` mit einem Argument schneidet das Array an einer bestimmten Stelle und gibt den entfernten Teil als Ergebnis zurück. Z.B. `splice(3)` bedeutet, dass wir das ursprüngliche Array vor dem `3`-ten Element zerschneiden und den entfernten Teil als Ergebnis zurückgeben.

   Wie oben erklärt findet das Zerschneiden bei einer positiven Zahl am (inklusive) `Array.length - n`-ten Element bis zum Ende. Wir zählen die Stellen also von rechts nach links. Wenn wir ein Minus davor setzen zählt `splice()` von rechts nach links. Statt `splice(Array.length - n)` können wir also auch schreiben `splice(-1 * n)`. Bei einer negativen Zahl am (inklusive) `Array.length - (Array.length + n)`-ten Element bis zum Ende des Arrays. Diesen Ausdruck können wir (für negative Zahlen) auch schreiben als `-n`. In beiden Fällen können wir also `splice(-1 * n)` schreiben.

2. Wir speichern den entfernten Array-Teil, um ihn später auf der anderen Seite des originalen Arrays anzuhängen.

3. Wir wollen die Zahlen aus dem entfernten Array-Teil nun am ursprünglichen Array anhängen. Wir nehmen die einzelnen Elemente des entfernten Array-Teils und hängen sie einzeln vorne am Ursprungsarray an. Wenn wir die Elemente aber vorne am Array anhängen wollen, können wir aber nicht das erste Element nehmen, sondern müssen mit dem letzten Element anfangen und uns zum ersten Element vorarbeiten. Ansonsten würden wir die Elemente in umgekehrter Reihenfolge anhängen. Daher drehen wir das Array mit Hilfe der `reverse()` Funktion um und lassen dann eine `forEach()` Funktion über das Array iterieren.

```
1 let rotate_array = function(arr, n) {
2   const rem = arr.splice(-1 * n);
3   rem.reverse().forEach(x => arr.unshift(x));
4
5   return arr;
6 };
```

Listing 7.15: My Javascript Example

Typische Fehler

- `unshift()` ineffizient

  Die Verwendung der Funktion `unshift()` ist ineffizient, v.a. bei langen Arrays. Denn die einzelnen Elemente müssen alle um einen Index nach

hinten verschoben werden. Bei kleinen Arrays ist die Performanceeinbuße vernachlässigbar. Bei langen Arrays ist es besser die `push()` Funktion zu verwenden.

- `reverse()` unnötig

  Die Tatsache, dass man beim Array vom letzten Element anfangen muss ist in diesem Fall richtig, da wir die Elemente von vorne anhängen (mit Hilfe von `unshift()`). Würden wir von hinten anhängen und eine Schleife benutzen, die vom ersten bis zum letzen Index läuft, bräuchten wir das Array nicht umzudrehen. Jedoch ist es hier unnötig das Array mit der `reverse()` Funktion umzukehren. Man kann auch die Schleife von hinten anfangen lassen. Einfach und effizienter wäre es eine `for` Schleife zu verwenden, die beim höchsten Index (also dem letzten Element) anfängt und sich dann vorarbeitet zum niedrigsten Index (also dem ersten Element).

Die obige Lösung liefert das korrekte Ergebnis. Ist bei langen Arrays jedoch ineffizient.

Lösung Durch die korrekte Verwendung der `splice()` Funktion und von ES6+ Syntax, den Rest/Spread Operator, können wir das Verschieben des Arrays intuitiv veranschaulichen.

Wir zerschneiden es an der `n`-ten Stelle von hinten. Und vertauschen den ausgeschnittenen Teil mit den Rest des ursprünglichen Arrays.

```
1 let rotate_array = function(arr, n) {
2   const rem = arr.splice(-1 * n);
3   return [...rem, ...arr];
4 };
```

Listing 7.16: My Javascript Example

7.5 Verschiebe alle 0 nach links

Aufgabe 7. *Gegeben ist ein Array. Verschiebe alle Elemente die gleich 0 sind nach links. Behalte dabei die Reihenfolge, der übrigen Zahlen bei.*
Beispiel: Gegeben ist folgender Array:

3	*23*	*0*	*-31*	*47*	*0*	*0*	*-65*	*0*	*69*

Das Fenster bewegt sich nun von links nach rechts und enthält folgende Zahlen:

1. *Schritt*

   Das Maximum ist wieder `69`

Stichwörter: Array, Suche, Sliding Window, Zeiger, Pointer, Dequeue

Vorgehensweise

1.

Listing 7.17: My Javascript Example

Typische Fehler

-

Lösung text....

Listing 7.18: My Javascript Example

7.6 Test

> **Aufgabe 8.** *Text*
> *Beispiel: Gegeben ist folgender Array mit einer Fensterbreite von 4.*
> | | | | | | |
> |---|---|---|---|---|---|
> | *3* | *23* | *-31* | *47* | *-65* | *69* |
>
> *Das Fenster bewegt sich nun von links nach rechts und enthält folgende Zahlen:*
>
> 1. *Schritt*
>
> *Das Maximum ist wieder* `69`

Stichwörter: Array, Suche, Sliding Window, Zeiger, Pointer, Dequeue

Vorgehensweise

1.

Listing 7.19: My Javascript Example

Typische Fehler

-

Lösung text....

Listing 7.20: My Javascript Example

Eine alternative Lösung ist es, einen `Heap()` zu verwenden.

```
1 // test...
```

Listing 7.21: My Javascript Example