

Inhalt

1	Algorithmen und Datenstrukturen in ES6+	3
1.1	Warum in ES6+	3
1.2	Behandelte Themen	3
1.3	Nicht behandelte Themen	3
2	Entwicklungsumgebung	5
3	JavaScript Paradigmen	7
3.1	Object-Orientiertes Programmieren	7
3.2	Funktionales Programmieren	7
4	JavaScript Konstrukte	9
4.1	Entscheidungskonstrukte	9
4.2	Wiederholungen	9
4.3	Funktionen	10
4.4	Scope	10
4.5	Closures	10
4.6	Hoisting	10
4.7	Call by Reference vs. Call by Value	10
4.8	Truthy und falsy	10
4.9	this	10
4.10	strict/sloppy mode	10
4.11	Number.EPSILON	10
5	Datenstrukturen	11
5.1	Array	11
5.2	Liste	20
5.3	Stack	25
5.4	Queue	26
5.5	Dequeue	27
5.6	Priority Queue	32
5.7	LinkedList	33
5.8	Zirkulare LinkedList	39
5.9	Zweifach verknüpfte LinkedList	39

5.10 Dictionary	39
5.11 Hashing	39
5.12 HashMap	39
5.13 MapTree	39
5.14 LinkedHashMap	39
5.15 Sets	39
5.16 Binäre Bäume	39
5.17 Graphen	39
5.18 AVL Tree	39
6 Algorithmen	41
6.1 Breitensuche	41
6.2 Tiefensuche	41
6.3 Bubble Sort	41
6.4 Selection Sort	41
6.5 Shellsort	41
6.6 Mergesort	41
6.7 Quicksort	41
6.8 Sequential Suche	41
6.9 Binäres Suchen	41
6.10 Suchen nach Minimum und Maximum	41
6.11 Rucksackproblem	41
6.12 Greedy Algorithm	41
7 Aufgaben	43
7.1 Binäres Suchen	43
7.2 Maximum im gleitenden Fenster	48

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 jedem 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

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 einem 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 Konstruktor 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 Konstruktor 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 Konstruktor 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 Konstruktor 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:

```
1 var myArray = [1, 2, 3];
2 myArray.push(4);
3 console.log(myArray);
4 // [1, 2, 3, 4]
5
6 myArray.unshift(0);
7 console.log(myArray);
8 // [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()`.

```
1 var myArray = [0, 1, 2, 3, 4];
2 console.log(myArray.pop());
3 // 4
4 console.log(myArray);
5 // [0, 1, 2, 3]
6
7 console.log(myArray.shift());
8 // 0
9 console.log(myArray);
10 // [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 Parameter die hinzuzufügenden Elemente.

```
1 var myArray = [1, 3, 4];
2 myArray.splice(1,0,2);
3 // adds a new element, 2, at index 1
4
5 myArray.splice(3,0,5,6,7);
6 // adds new elements, 5,6, and 7, at index 3
7
8 myArray.splice(5, 1);
9 // removes one element at index 5
10
11 myArray.splice(2, 3);
12 // removes 3 elements starting at index 2
13
14 myArray.splice(3, 1, 99);
15 // replaces one element at index 3 with the new element 99
```

```

16 myArray.splice(3, 2, 42);
17 // replaces 2 elements starting at index 3 with the new element 42
18
19 myArray.splice(999,0,2);
20 // if the index (first parameter) is larger than the array length,
21 // then it will just adds a new element to the end

```

Listing 5.8: 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 5.9: 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 5.10: 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 5.11: 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 5.12: 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 5.13: 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 5.14: 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]]
8
9 var myArray = [1, 2, 3, [4, [5, 6]]];
10 console.log(myArray.flat(2));
11 // [1, 2, 3, 4, 5, 6]

```

Listing 5.15: 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());

```

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


```
3 // [1, 2, 3, 6]
```

Listing 5.16: 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 anschließend 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 5.17: 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 5.18: 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 5.19: 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 5.20: Array Konstruktor

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

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 5.21: 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 5.22: 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 5.23: Array Konstruktor

Es muss aber hier erwähnt werden, 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 5.24: 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 einzelne 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 5.25: 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 `falsey` 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
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 5.26: 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.

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

```

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 5.27: Array Konstruktor

Will man noch dazu herauszufinden, an welcher Position das gesuchte Element sich befindet, gibt es `indexOf()`. Diese nimmt 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 5.28: 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 5.29: 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 5.30: 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.

```

1 class List {
2     constructor(items = []) {
3         this.store = items;
4         this.pos = 0;
5         this.size = items.length;
6     }
7     clear() { /*...*/ }
8     toString() { /*...*/ }
9     get(index) { /*...*/ }
10    indexOf (item) { /*...*/ }
11    lastIndexOf () { /*...*/ }
12    isEmpty () { /*...*/ }
13    insert(index) { /*...*/ }
14    add() { /*...*/ }
15    remove() { /*...*/ }
16    front() { /*...*/ }
17    end() { /*...*/ }
18    prev() { /*...*/ }
19    next() { /*...*/ }
20    moveTo(index) { /*...*/ }
21    sublist(from, to) { /*...*/ }
22 }

```

Listing 5.31: Array Konstruktor

Die einzelnen 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.

```

1 clear() {
2     this.store = [];
3     this.size = this.pos = 0;
4 }

```

Listing 5.32: Array Konstruktor

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

```

1 toString() {
2     return this.store;
3 }

```

Listing 5.33: 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 bei der Index dieser Zahl entspricht.

```
1 get(index = this.pos) {  
2   return this.store[index];  
3 }
```

Listing 5.34: 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.

```
1 indexOf(item) {  
2   return this.store.indexOf(item);  
3 }
```

Listing 5.35: 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.

```
1 lastIndexOf(item) {  
2   return this.store.lastIndexOf(item);  
3 }
```

Listing 5.36: Array Konstruktor

isEmpty() - Liefert true, wenn die Liste leer ist

```
1 isEmpty() {  
2   return this.store.length === 0;  
3 }
```

Listing 5.37: 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) - Fügt ein Element an einer bestimmten Stelle ein

```
1 insert(el, index) {  
2   this.store.splice(index, 0, el);  
3 }
```

Listing 5.38: Array Konstruktor

add(el) - Fügt ein Element am Ende der Liste

```
1 add(el) {  
2   this.store.push(el);  
3 }
```

Listing 5.39: Array Konstruktor

remove(index) - Entfernt ein Element an einer bestimmten Stelle

```
1 remove(index) {  
2   this.store.splice(index, 1);  
3 }
```

Listing 5.40: Array Konstruktor

front() - Setzt die Position am Listenanfang

```
1 remove(index) {  
2   this.pos = 0;  
3 }
```

Listing 5.41: Array Konstruktor

end() - Setzt die Position am Listenende

```
1 end() {  
2   this.pos = this.store.length - 1;  
3 }
```

Listing 5.42: Array Konstruktor

prev() - Setzt die Position um eins nach vorne

```
1 prev() {  
2   if (this.pos === 0) {  
3     return 0;  
4   }  
5   return --this.pos;  
6 }
```

Listing 5.43: Array Konstruktor

next() - Setzt die Position um eins nach hinten

```
1 next() {  
2   if (this.pos === this.store.length - 1) {  
3     return this.pos;  
4   }  
5   return ++this.pos;  
6 }
```

Listing 5.44: Array Konstruktor

moveTo(index) - Setzt die Position an einer bestimmten Stelle

```

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

```

Listing 5.45: 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 sublist 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 5.46: 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 Funktionen 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

`DequejEj` 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 `DequejEj` 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 `DequejEj` 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 `DequejEj` 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 `CollectionjEj` Specified by: `contains` in interface `DequejEj` Specified by: `contains` in interface `ListjEj` Overrides: `contains` in class `AbstractCollectionjEj` 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 `CollectionjEj` Specified by: `size` in interface `DequejEj` Specified by: `size` in interface `ListjEj` Specified by: `size` in class `AbstractCollectionjEj` 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 `CollectionjEj` Specified by: `add` in interface `DequejEj` Specified by: `add` in interface `ListjEj` Specified by: `add` in interface `QueuejEj` Overrides: `add` in class `AbstractListjEj` 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 `CollectionjEj` Specified by: `remove` in interface `DequejEj` Specified by: `remove` in interface `ListjEj` Overrides: `remove` in class `AbstractCollectionjEj` Parameters: `o` - element to be removed from this list, if present Returns: true if this list contained the specified element `addAll` public boolean `addAll(Collectionj? extends Ej 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 `CollectionjEj` Specified by: `addAll` in interface `ListjEj` Overrides: `addAll` in class `AbstractCollectionjEj` 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 oc-

currence 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 $(o == null ? \text{get}(i) == null : o.equals(\text{get}(i)))$, 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

Typische Fehler

```
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.slice(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 beginnt 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. `currentIndex` (`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 versuchen folgende Fälle zu berücksichtigen:

- Element, das

Ermittelt den Wert in der Mitte des Arrays

2. Teile ihn dann in zwei Arrays auf
3. Suche in jeweils der zwei Subarrays nach der Zahl in Binary Search

Typische Fehler

```

1 let binarySearchSwitched = function(arr, key) {
2   const maxIndex = arr.length - 1;
3   let startPointer = 0;
4   let endPointer = maxIndex;
5   while (startPointer <= endPointer) {
6     const currentPointer = (startPointer + endPointer) / 2 | 0;
7     const oldCurrentKey = arr[currentPointer];
8
9     if (oldCurrentKey < key) {
10      // go to right
11      startPointer = currentPointer + 1;
12      const newCurrentPointer = (startPointer + endPointer) / 2;
13      const newCurrentKey = arr[newCurrentPointer];
14      if (newCurrentKey < oldCurrentKey) {
15        // we are on the right; newCurrentKey has to be bigger than
16        // oldCurrentKey
17        // if it is not, then we have to look at the other side
18        return searchBinay(0, currentPointer - 1);
19      }
20    } else if (oldCurrentKey > key) {
21      // go to left
22      endPointer = currentPointer - 1;
23      const newCurrentPointer = (startPointer + endPointer) / 2;
24      const newCurrentKey = arr[newCurrentPointer];
25      if (newCurrentKey > oldCurrentKey) {
26        // we are on the left; newCurrentKey has to be smaller than
27        // oldCurrentKey
28        // if it is not, then we have to look at the other side
29        return searchBinay(currentPointer + 1, maxIndex);
30      }
31    } else {
32      return currentPointer;
33    }
34  }
35 }

```

```

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

- Sehr lang, sehr kompliziert und zu spezifisch

Die Länge des Codes ist geschuldet durch die Tatsache, dass im Code spezifisch auf 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) eingegangen wird und versucht wird diese abzufangen.

- Nutzt vorhandene Informationen nicht

Offensichtlich ist es wichtig zu wissen, wo der Schnitt sich befindet bzw. welche Elemente vor dem Schnitt sich befindet. Man hätte ausnutzen können, dass die Zahlen sortiert sind und dadurch die Elemente, die Dies wird aber nicht berücksichtigt. Obiger Code liefert für folgendes Array 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

- Kein early exit

Es wurde nicht versucht einen early exit herbeizuführen, also eine Bedingung zu suchen bei der das Programm frühzeitig mit dem richtigen Ergebnis terminiert.

Lösung Die Laufzeit
Zeit komplexität

```

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[
16    end]) {
17    return binary_search_recs(arr, mid + 1, end, key);
18  } else if (arr[end] <= arr[mid]) {
19    return binary_search_recs(arr, mid + 1, end, key);
20  } else if (arr[st] >= arr[mid]) {
21    return binary_search_recs(arr, st, mid - 1, key);
22  }
23  return -1;
24 };
25
26 let binarySearchSwitched = function(arr, key) {
27   return binarySearchRecursive(arr, 0, arr.length - 1, key);
28 };

```

Listing 7.4: My Javascript Example

7.2 Maximum im gleitenden Fenster

Aufgabe 3. 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 durch 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

```

1 const findMaxSlidingSindow = function(arr, window_size) {
2   const getMax = (startWindow, endWindow) => {
3     const max = arr
4       .slice(startWindow, endWindow + 1)
5       .reduce((max, item) => item > max ? item : max, -Infinity);
6     result.push(max);
7   }
8   const result = [];
9   let startWindow = 0;
10  let endWindow = window_size - 1;
11  let max =
12  while (endWindow < arr.length) {
13
14    getMax(startWindow++, endWindow++);
15  }
16  return result;
17 };

```

Listing 7.5: 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. Beispiw. 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 `Deque` 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 einmal durch die Zahlen im Array gehen und dabei das Maximum speichern. Nachfolgend werde ich begründen, warum eine `Deque` 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 `Deque` 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 `Deque`. 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 folgt aussehen:

1. Schritt

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

Erste Zahl ist 7. Weil die `Deque` noch leer ist, pushen wir das erste Index sie in unsere `Deque` 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 `Deque` die Zahlen der kleiner nach speichern) als das aktuelle max und wir pushen sie ins

`Deque`:

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 `Deque`

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 `Dequeue`

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 {
35     dequeue.shift();
36 }
37 dequeue.push(i);
38 result.push(arr[dequeue[0]]);
39 }
40
41 return result;
42 };
```

Listing 7.6: My Javascript Example

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

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

Listing 7.7: My Javascript Example