

# 1

## Motivation

*Algorithmen & Datenstrukturen · Sommersemester 2026*

Prof. Dr. Sebastian Wild

# Outline

## 1 Motivation

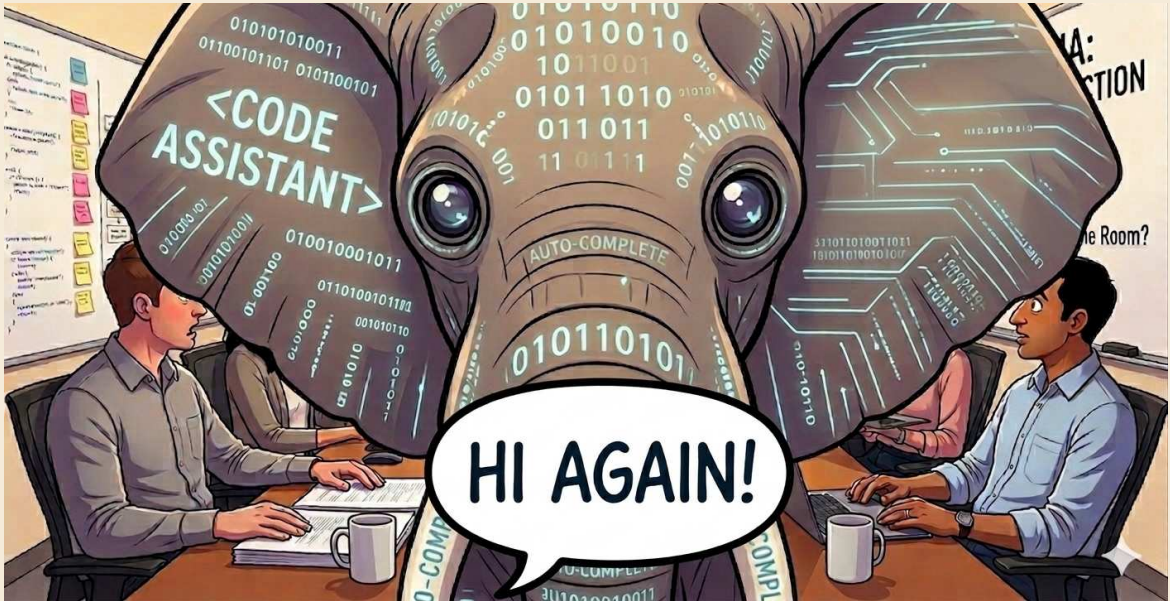
- 1.1 Ketzerische Fragen
- 1.2 Effizienz in Anwendungen
- 1.3 Abstraktion
- 1.4 Die Sprache der Experten

# Wozu Algorithmen und Datenstrukturen?



## **1.1 Ketzerische Fragen**

# Kann das nicht die KI machen?



## Kann das nicht die KI machen?

- ▶ Ja und Nein.
- ▶ Mit präzisen und passenden Anweisungen kann KI Code generieren.
- ↪ Genau hier profitiert man aber davon, die richtigen Abstraktionen zu verwenden!  
**Dafür muss man wissen, wovon man redet.**

# Kann das nicht die KI machen?

- ▶ Ja und Nein.
- ▶ Mit präzisen und passenden Anweisungen kann KI Code generieren.
- ↪ Genau hier profitiert man aber davon, die richtigen Abstraktionen zu verwenden!  
**Dafür muss man wissen, wovon man redet.**
- ▶ KI liefert die häufigste Lösung (aus Training), die ungefähr passt.  
*Nur weil ein Ansatz in Medium-Blogs am häufigsten vorkommen ist er nicht unbedingt korrekt oder effizient!*
- ↪ **Für die Unterscheidung brauchen Sie das ADS-Wissen!**
- ▶ Was tun bei *Performance Leaks*?

# Nicht nur mein Ratschlag

*“Master the fundamentals first. This is critical, because you can’t evaluate code quality without understanding what good architecture looks like. You can’t write effective tests without knowing what can break. You can’t catch hallucinations without domain expertise.*

*It’s important for experienced developers to keep these skills from atrophying, and junior developers to build them in the first place.”*

*Eira May, Mind the gap: Closing the AI trust gap for developers, [stackoverflow.blog/2026/02/18/closing-the-developer-ai-trust-gap](https://stackoverflow.blog/2026/02/18/closing-the-developer-ai-trust-gap)*

# Nicht nur mein Ratschlag

*“Master the fundamentals first. This is critical, because you can’t evaluate code quality without understanding what good architecture looks like. You can’t write effective tests without knowing what can break. You can’t catch hallucinations without domain expertise.*

*It’s important for experienced developers to keep these skills from atrophying, and junior developers to build them in the first place.”*

*Eira May, Mind the gap: Closing the AI trust gap for developers, [stackoverflow.blog/2026/02/18/closing-the-developer-ai-trust-gap](https://stackoverflow.blog/2026/02/18/closing-the-developer-ai-trust-gap)*

*“What you want to get to, particularly as your career evolves, is mastery. That’s how you kind of escape the thing that everybody can do and get more differentiation. The concern I have is this culture of, ‘Well, I’m not even going to try to understand what’s going on. I’m just going to spend some tokens, and maybe it’ll be great.’*

*[...]*

*Getting us out of writing boilerplate, getting us out of memorizing APIs, getting us out of looking up that thing from Stack Overflow; I think this is really profound. This is a good use. The thing that I get **concerned** about is if you go so far as to not care about what you’re looking up on Stack Overflow and why it works that way and **not learning from it.**”*

*Chris Lattner, [fast.ai/posts/2025-10-30-build-to-last.html](https://fast.ai/posts/2025-10-30-build-to-last.html)*

# Kann ich nicht einfach Unit Testen und gut?

- ▶ (nicht für sicherheitsrelevante Software)
- ▶ Unit Tests prüfen **Korrektheit**
- ▶ Skalierbarkeit kann man so nicht testen!

## Von 99% zu 100%

- ▶ Zwischen “fast komplett richtig” und “100% korrekt” liegen oft Welten

## Von 99% zu 100%

- ▶ Zwischen “fast komplett richtig” und “100% korrekt” liegen oft Welten
- ▶ *Hierarchische Abstraktion* und Blackbox-Reuse sind omnipräsent in der Informatik
  - ▶ “algorithmic thinking”
  - ▶ Problemlösestrategien

## Von 99% zu 100%

- ▶ Zwischen “fast komplett richtig” und “100% korrekt” liegen oft Welten
- ▶ **Hierarchische Abstraktion** und Blackbox-Reuse sind omnipräsent in der Informatik
  - ▶ “algorithmic thinking”
  - ▶ Problemlösestrategien

*unsere einzige erwiesenermaßen effektive Waffe für wachsende Komplexität von Systemen*

↪ kleinste Fehler in der Realisierung propagieren sich über viele Abstraktionsschichten hinweg zu Systemversagen

- ▶ *Blackbox-Reuse braucht pedantische Korrektheit*

## **1.2 Effizienz in Anwendungen**

# Skalierbarkeit

## Analogie: Architektur

- ▶ 1m Brücke: Holzplanke genügt!
- ▶ 10x  $\rightsquigarrow$  Konstruktion nötig
- ▶ 1000x  $\rightsquigarrow$  Meisterleistung der Bauingenieurskunst

*Eine 1km Brücke ist nicht einfach ein 1000-fach größeres Brett.*

# Skalierbarkeit

## Analogie: Architektur

- ▶ 1m Brücke: Holzplanke genügt!
- ▶ 10x  $\rightsquigarrow$  Konstruktion nötig
- ▶ 1000x  $\rightsquigarrow$  Meisterleistung der Bauingenieurskunst

*Eine 1km Brücke ist nicht einfach ein 1000-fach größeres Brett.*



# Skalierbarkeit

## Analogie: Architektur

- ▶ 1m Brücke: Holzplanke genügt!
- ▶ 10x  $\rightsquigarrow$  Konstruktion nötig
- ▶ 1000x  $\rightsquigarrow$  Meisterleistung der Bauingenieurskunst

*Eine 1km Brücke ist nicht einfach ein 1000-fach größeres Brett.*



2m

$\sim 10x$   
 $\longrightarrow$



12m

$\sim 10x$   
 $\longrightarrow$



170m

$\sim 10x$   
 $\longrightarrow$



1280m

# Skalierbarkeit

## Analogie: Architektur

- ▶ 1m Brücke: Holzplanke genügt!
- ▶ 10x  $\rightsquigarrow$  Konstruktion nötig
- ▶ 1000x  $\rightsquigarrow$  Meisterleistung der Bauingenieurskunst

*Eine 1km Brücke ist nicht einfach ein 1000-fach größeres Brett.*



2m

€ 50 000

$\sim 10x$   
→

$\sim 10x$   
→



12m

€ 350 000

$\sim 10x$   
→

$\sim 50x$   
→

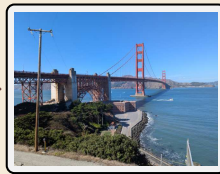


170m

€ 22 000 000

$\sim 10x$   
→

$\sim 200x$   
→



1280m

€ 4 500 000 000

(Kosten sind Schätzungen der heutigen Kosten)

# Skalierbarkeit

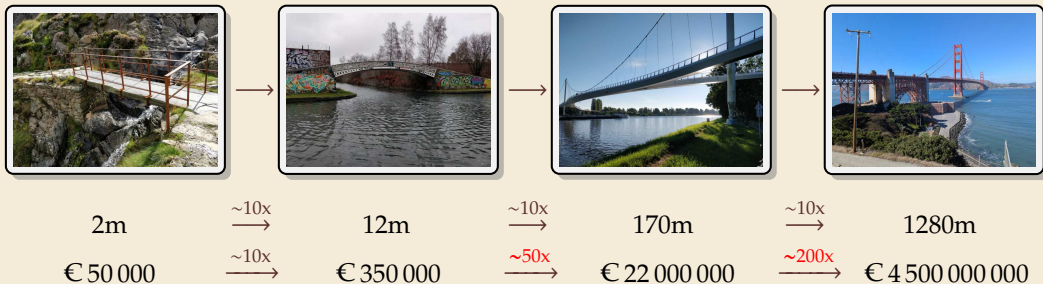
## Analogie: Architektur

- ▶ 1m Brücke: Holzplanke genügt!
- ▶ 10x  $\rightsquigarrow$  Konstruktion nötig
- ▶ 1000x  $\rightsquigarrow$  Meisterleistung der Bauingenieurskunst

## Informatik

- ▶ Eingabegrößen in Informatiksystemen erreichen oft Faktor 1 000 000 000!
- 1 Byte  $\rightarrow$  1 GB
- $\rightsquigarrow$  Können uns keine Holzbretter leisten.

*Eine 1km Brücke ist nicht einfach ein 1000-fach größeres Brett.*



(Kosten sind Schätzungen der heutigen Kosten)

## Gibt's das nicht als fertige Library?

- ▶ klassische Datenstrukturen (wie die in ADS!) gibt es in der Regel als Software Library
- ↪ nicht das Rad neu erfinden, wenn es nicht nötig ist!
- ▶ **Aber:** Diese Bibliotheken rein als "*magic black boxes*" zu begreifen ist unzureichend.
    - ▶ müssen Performancecharakteristika auf Anwendung anpassen
    - ▶ Fehlersuche bei "Performance leaks"
    - ▶ Erweiterungen

## Gibt's das nicht als fertige Library?

- ▶ klassische Datenstrukturen (wie die in ADS!) gibt es in der Regel als Software Library
- ↪ nicht das Rad neu erfinden, wenn es nicht nötig ist!
- ▶ **Aber:** Diese Bibliotheken rein als "*magic black boxes*" zu begreifen ist unzureichend.
    - ▶ müssen Performancecharakteristika auf Anwendung anpassen
    - ▶ Fehlersuche bei "Performance leaks"
    - ▶ Erweiterungen

Wissen um Performancecharakteristika ist Ihr *Bullsh\*t Detector*

## Gibt's das nicht als fertige Library?

- ▶ klassische Datenstrukturen (wie die in ADS!) gibt es in der Regel als Software Library
- ↪ nicht das Rad neu erfinden, wenn es nicht nötig ist!
- ▶ **Aber:** Diese Bibliotheken rein als *“magic black boxes”* zu begreifen ist unzureichend.
  - ▶ müssen Performancecharakteristika auf Anwendung anpassen
  - ▶ Fehlersuche bei *“Performance leaks”*
  - ▶ Erweiterungen

Wissen um Performancecharakteristika ist Ihr *Bullsh\*t Detector*

*“I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important.*

*Bad programmers worry about the code.*

*Good programmers worry about data structures and their relationships.”*

*Linus Torvalds*

# „Effizienz“ ist kontextabhängig



## 1.3 Abstraktion

# Power of Abstraction

Sie erinnern sich an eben:

- ▶ **Hierarchische Abstraktion** und Blackbox-Reuse sind omnipräsent in der Informatik
  - ▶ “algorithmic thinking”
  - ▶ Problemlösestrategien

*unsere einzige erwiesenermaßen effektive Waffe für wachsende Komplexität von Systemen*

# Power of Abstraction

Sie erinnern sich an eben:

- ▶ **Hierarchische Abstraktion** und Blackbox-Reuse sind omnipräsent in der Informatik
  - ▶ "algorithmic thinking"
  - ▶ Problemlösestrategien

*unsere einzige erwiesenermaßen effektive Waffe für wachsende Komplexität von Systemen*

- ▶ typische Beispiele: Dateisysteme, TCP-Protokoll, String-Funktionen in Java, ...
  - ▶ Oft sind diese Abstraktionen gerade Algorithmen und Datenstrukturen!

char {}

UT7-16

# Power of Abstraction

Sie erinnern sich an eben:

- ▶ **Hierarchische Abstraktion** und Blackbox-Reuse sind omnipräsent in der Informatik
  - ▶ “algorithmic thinking”
  - ▶ Problemlösestrategien

*unsere einzige erwiesenermaßen effektive Waffe für wachsende Komplexität von Systemen*

- ▶ typische Beispiele: Dateisysteme, TCP-Protokoll, String-Funktionen in Java, ...
  - ▶ Oft sind diese Abstraktionen gerade Algorithmen und Datenstrukturen!
- ▶ Algorithmen sind der ultimative **wiederverwendbare Code**
  - ▶ quasi per Definition ein wiederkehrendes Problem, das wir ein für alle mal lösen
- ▶ Abstrakte Datentyp<sup>∧</sup>e (ADTs) sind die ultimative Form von semantischer Kapselung (*encapsulation and information hiding*)
  - ▶ Datenstrukturen als austauschbare Implementierungen entkoppeln Realisierung von Verwendung

## Unter die Motorhaube

- ▶ Wenn die Abstraktion perfekt passt, reicht die Library-Implementierung
- ▶ aber Achtung: früher oder später muss man in die Blackbox reinschauen

*“All non-trivial abstractions, to some degree, are leaky.”*

*Joel Spolsky, [joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions](http://joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions)*

*Leaky abstractions*

# Unter die Motorhaube

- ▶ Wenn die Abstraktion perfekt passt, reicht die Library-Implementierung
- ▶ aber Achtung: früher oder später muss man in die Blackbox reinschauen

*“All non-trivial abstractions, to some degree, are leaky.”*

*Joel Spolsky, [joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions](http://joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions)*

- ▶ “Autofahrer oder Mechaniker”

↪ keine Autos mit  
**zugeschweißter Motorhaube** bauen



## Ratschlag aus dem Jahre 2002

*“The law of leaky abstractions means that whenever somebody comes up with a **wizzy new code-generation tool** that is supposed to make us all ever-so-efficient, you hear a lot of people saying ‘learn how to do it manually first, then use the wizzy tool to save time.’ **Code generation tools which pretend to abstract out something, like all abstractions, leak, and the only way to deal with the leaks competently is to learn about how the abstractions work and what they are abstracting.***

*So the abstractions save us time working, but they don't save us time learning.*

*And all this means that paradoxically, even as we have higher and higher level programming tools with better and better abstractions, becoming a proficient programmer is getting harder and harder.”*

*Joel Spolsky, [joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions](http://joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions)*

## **1.4 Die Sprache der Experten**

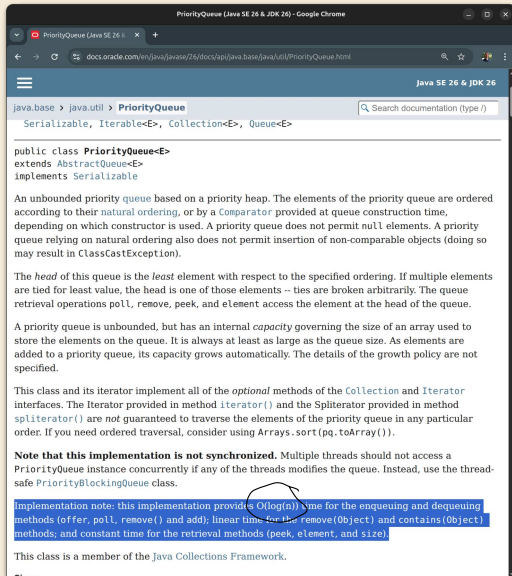
# Abstraktion wird Vokabel

- ▶ Unter Experten ersetzen einzelne Begriffe ganze komplexe Themenfelder

↪ wiederverwertbares *Verständnis*

- ▶ Es wird erwartet, dass Sie damit zurechtkommen!

# Beispiele aus der Java API



The screenshot shows the Java API documentation for `PriorityQueue` in a browser window. The page title is "PriorityQueue (Java SE 26 & JDK 26) - Google Chrome". The URL is `docs.oracle.com/en/java/javase/26/docs/api/java.base/java.util/PriorityQueue.html`. The breadcrumb navigation shows `java.base > java.util > PriorityQueue`. The class is listed as `Serializable, Iterable<E>, Collection<E>, Queue<E>`. The class signature is `public class PriorityQueue<E>`, which `extends AbstractQueue<E>` and `implements Serializable`. The documentation describes it as an unbounded priority queue based on a priority heap, ordered by natural ordering or a `Comparator`. It notes that the `head` is the *least* element and that the queue is unbounded but has an internal *capacity*. It also states that the class and its iterator implement *optional* methods of `Collection` and `Iterator`. A **Note** states that the implementation is **not synchronized**. A blue highlight covers the implementation note, with a red circle around the `O(log(n))` complexity for enqueueing and dequeuing methods.

PriorityQueue (Java SE 26 & JDK 26) - Google Chrome

PriorityQueue (Java SE 26) | +

docs.oracle.com/en/java/javase/26/docs/api/java.base/java.util/PriorityQueue.html

Java SE 26 & JDK 26

java.base > java.util > PriorityQueue

Search documentation (type /)

Serializable, Iterable<E>, Collection<E>, Queue<E>

---

public class **PriorityQueue<E>**  
extends AbstractQueue<E>  
implements Serializable

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a `Comparator` provided at queue construction time, depending on which constructor is used. A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in `ClassCastException`).

The *head* of this queue is the *least* element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily. The queue retrieval operations `poll`, `remove`, `peek`, and `element` access the element at the head of the queue.

A priority queue is unbounded, but has an internal *capacity* governing the size of an array used to store the elements on the queue. It is always at least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The details of the growth policy are not specified.

This class and its iterator implement all of the *optional* methods of the `Collection` and `Iterator` interfaces. The `Iterator` provided in method `iterator()` and the `Splititerator` provided in method `splititerator()` are *not* guaranteed to traverse the elements of the priority queue in any particular order. If you need ordered traversal, consider using `Arrays.sort(pq.toArray())`.

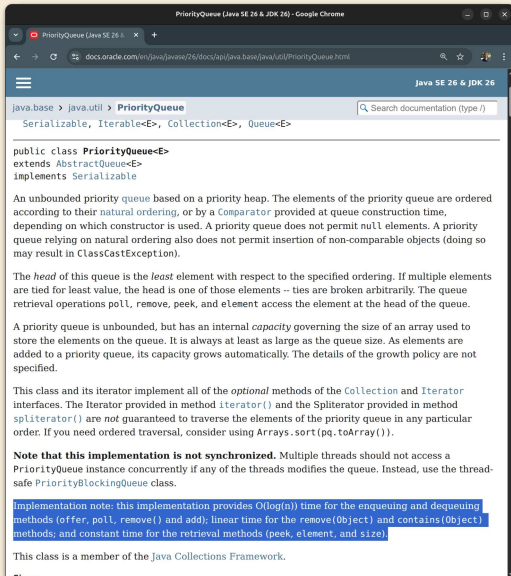
**Note that this implementation is not synchronized.** Multiple threads should not access a `PriorityQueue` instance concurrently if any of the threads modifies the queue. Instead, use the thread-safe `PriorityBlockingQueue` class.

Implementation note: this implementation provides  $O(\log(n))$  time for the enqueueing and dequeuing methods (`offer`, `poll`, `remove()` and `add`); linear time for the `remove(Object)` and `contains(Object)` methods; and constant time for the retrieval methods (`peek`, `element`, and `size`).

This class is a member of the Java Collections Framework.

Class:

# Beispiele aus der Java API



PriorityQueue (Java SE 26 & JDK 26) - Google Chrome

docs.oracle.com/en/java/javase/26/docs/api/java.base/java/util/PriorityQueue.html

java.base > java.util > **PriorityQueue**

Serializable, Iterable<E>, Collection<E>, Queue<E>

```
public class PriorityQueue<E>
  extends AbstractQueue<E>
  implements Serializable
```

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a `Comparator` provided at queue construction time, depending on which constructor is used. A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in `ClassCastException`).

The *head* of this queue is the *least* element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily. The queue retrieval operations `poll`, `remove`, `peek`, and `element` access the element at the head of the queue.

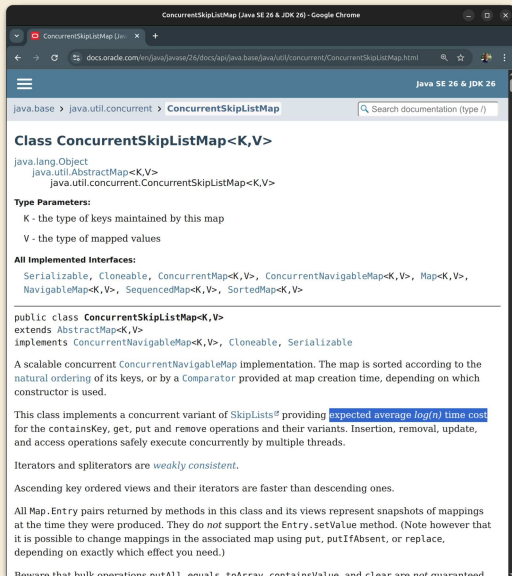
A priority queue is unbounded, but has an internal *capacity* governing the size of an array used to store the elements on the queue. It is always at least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The details of the growth policy are not specified.

This class and its iterator implement all of the *optional* methods of the `Collection` and `Iterator` interfaces. The iterator provided in method `iterator()` and the `Spliterator` provided in method `spliterator()` are *not* guaranteed to traverse the elements of the priority queue in any particular order. If you need ordered traversal, consider using `Arrays.sort(pq.toArray())`.

**Note that this implementation is not synchronized.** Multiple threads should not access a `PriorityQueue` instance concurrently if any of the threads modifies the queue. Instead, use the thread-safe `PriorityBlockingQueue` class.

**Implementation note:** this implementation provides  $O(\log(n))$  time for the enqueueing and dequeuing methods (`offer`, `poll`, `remove()` and `add`); linear time for the `remove(Object)` and `contains(Object)` methods; and constant time for the retrieval methods (`peek`, `element`, and `size`).

This class is a member of the Java Collections Framework.



ConcurrentSkipListMap (Java SE 26 & JDK 26) - Google Chrome

docs.oracle.com/en/java/javase/26/docs/api/java.base/java/util/concurrent/ConcurrentSkipListMap.html

java.base > java.util.concurrent > **ConcurrentSkipListMap**

### Class ConcurrentSkipListMap<K,V>

```
java.lang.Object
  java.util.AbstractMap<K,V>
    java.util.concurrent.ConcurrentSkipListMap<K,V>
```

**Type Parameters:**

- K - the type of keys maintained by this map
- V - the type of mapped values

**All Implemented Interfaces:**

Serializable, Cloneable, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, Map<K,V>, NavigableMap<K,V>, SequencedMap<K,V>, SortedMap<K,V>

```
public class ConcurrentSkipListMap<K,V>
  extends AbstractMap<K,V>
  implements ConcurrentNavigableMap<K,V>, Cloneable, Serializable
```

A scalable concurrent `ConcurrentNavigableMap` implementation. The map is sorted according to the natural ordering of its keys, or by a `Comparator` provided at map creation time, depending on which constructor is used.

This class implements a concurrent variant of `SkipLists`<sup>®</sup> providing **expected average  $\log(n)$  time cost** for the `containsKey`, `get`, `put` and `remove` operations and their variants. Insertion, removal, update, and access operations safely execute concurrently by multiple threads.

Iterators and spliterators are *weakly consistent*.

Ascending key ordered views and their iterators are faster than descending ones.

All `Map.Entry` pairs returned by methods in this class and its views represent snapshots of mappings at the time they were produced. They do *not* support the `Entry.setValue` method. (Note however that it is possible to change mappings in the associated map using `put`, `putIfAbsent`, or `replace`, depending on exactly which effect you need.)

**Beware that bulk operations `putAll`, `equalsToArray`, `containsValue`, and `clear` are not guaranteed**

# Gatekeeper: Coding Interviews

*Immer noch nicht überzeugt, dass ADS für Sie wertvoll ist?*

# Gatekeeper: Coding Interviews

*Immer noch nicht überzeugt, dass ADS für Sie wertvoll ist?*

Für die meisten Karrieren in der Tech Industrie sind *technische Interviews* Standard.

- ▶ Fragen zu Algorithmen und Datenstrukturen (sowie ihrer Implementierung) sind die häufigsten Themen
- ▶ Standard-Kanon an Universitäten
- ▶ gut testbar
- ▶ auch allen Interviewern bekannt!



[xkcd.com/2483/](http://xkcd.com/2483/)

# Zusammenfassung

## Ziele von ADS:

1. **Baukasten** aus Algorithmen und Datenstrukturen aufbauen
  - ▶ *best practices*: Liste von A. & DS. für Standardprobleme
  - ▶ intellektuelle Errungenschaften der Informatik
  - ▶ Anpassungen und Erweiterungen ermöglichen

# Zusammenfassung

## Ziele von ADS:

1. **Baukasten** aus Algorithmen und Datenstrukturen aufbauen
  - ▶ *best practices*: Liste von A. & DS. für Standardprobleme
  - ▶ intellektuelle Errungenschaften der Informatik
  - ▶ Anpassungen und Erweiterungen ermöglichen
2. Algorithmen & Datenstrukturen **bewerten** können

# Zusammenfassung

## Ziele von ADS:

1. **Baukasten** aus Algorithmen und Datenstrukturen aufbauen
  - ▶ *best practices*: Liste von A. & DS. für Standardprobleme
  - ▶ intellektuelle Errungenschaften der Informatik
  - ▶ Anpassungen und Erweiterungen ermöglichen
2. Algorithmen & Datenstrukturen **bewerten** können
3. mit anderen Experten über Algorithmen & Datenstrukturen **reden** können
  - ▶ Vokabular aufbauen (↔ Baukasten)
  - ▶ präzise Beschreibungen (von Performance) verstehen und erstellen
  - ▶ erkennen, wenn jemand „Mist“ erzählt