

# Algorithmen 1

## Listen und binäre Suche



# Wiederholung: Zwei Arten der Datenablage

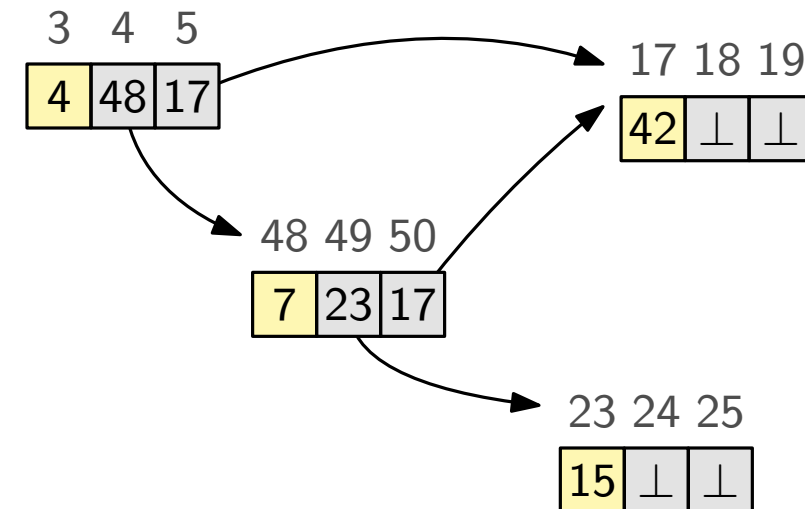
## Felder (Arrays)

- Menge aufeinanderfolgender Speicherzellen
- Zugriff mit Adresse bzw. Index an beliebiger Stelle in  $\Theta(1)$
- sehr nah an der Hardware
- letztes Mal: Komfort-Funktionen für dynamisches Wachstum

Index:	0	1	2	3	4	5	6	7	8	9	10	11
Adresse:	30	31	32	33	34	35	36	37	38	39	40	41
Daten:	4	48	89	1	0	9	13	7	32	76	17	5

## Verzeigte Strukturen

- viele kleine Stückchen Speicher (Knoten)
- ein Knoten speichert:
  - Daten, die uns tatsächlich interessieren
  - Speicheradressen anderer Knoten (Zeiger)
- Zugriff durch Navigation entlang Zeiger
- abstrahiert stärker von der Hardware
- heute: Listen als einfaches Beispiel



# Doppelt verkettete Liste

## Ziele

- speichere eine Folge von Zahlen  $\langle 4, 48, 89, 1 \rangle$
- flexible Einfüge- und Löschoperationen

(abstraktes Objekt, **Mathe**)  
 (Funktionalität, **Softwaretechnik**)

## Repräsentation und Effiziente Umsetzung

(**Algorithmik**)

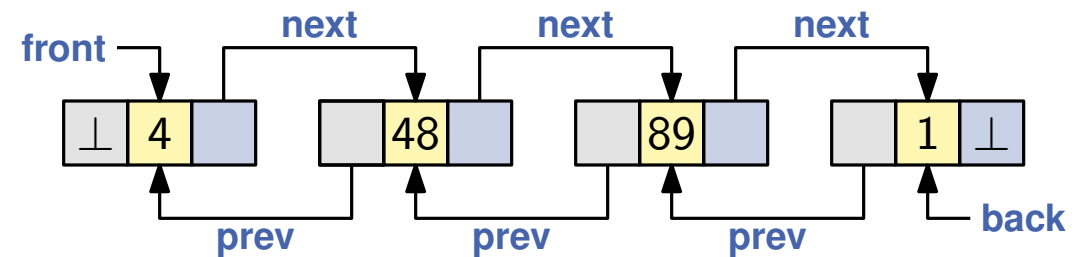
- jeder Knoten der Datenstruktur speichert:

- eine der Zahlen der Folge **value**
- Zeiger zum nächsten Knoten **next**
- Zeiger zum vorherigen Knoten **prev**

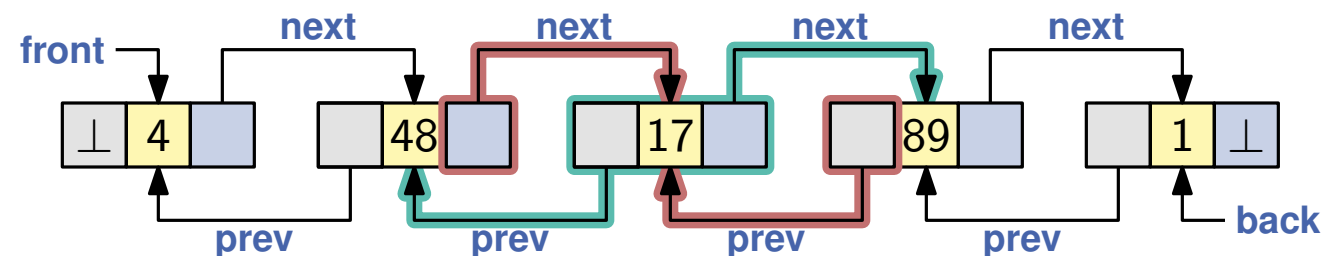
- Einstiegspunkte: **front**, **back**

- Einfüge- und Löschoperationen

- Änderungen sind nur lokal
- konstant viele Zeiger umhängen  
 → konstante Laufzeit ( $\Theta(1)$ )



**Beispiel:**  $\langle 4, 48, \overset{17}{89}, 1 \rangle$



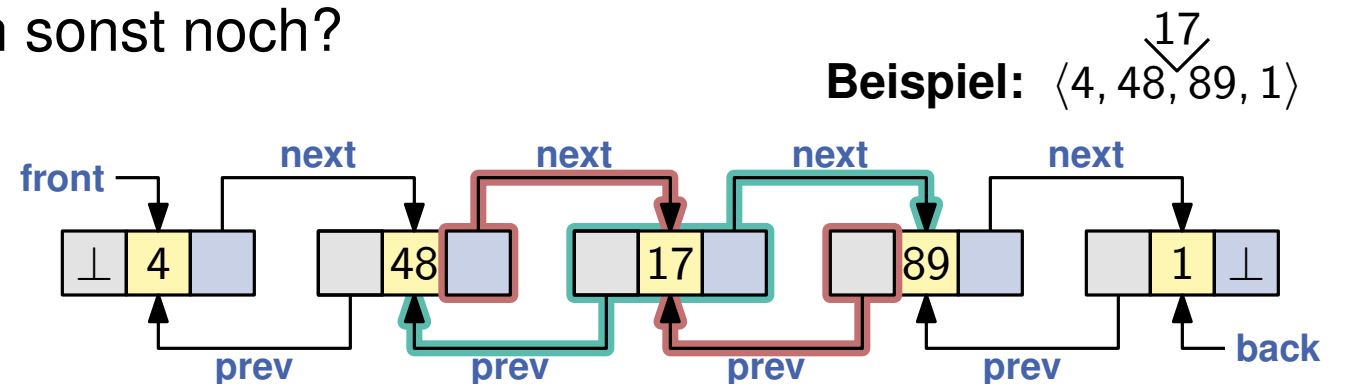
# Wo stehen wir?

## Gerade gesehen

- grundlegende Funktionsweise einer Liste
- High-Level Verständnis, warum flexibles Einfügen und Löschen effizient geht
- → hohes Abstraktionslevel

## Offene Detailfragen

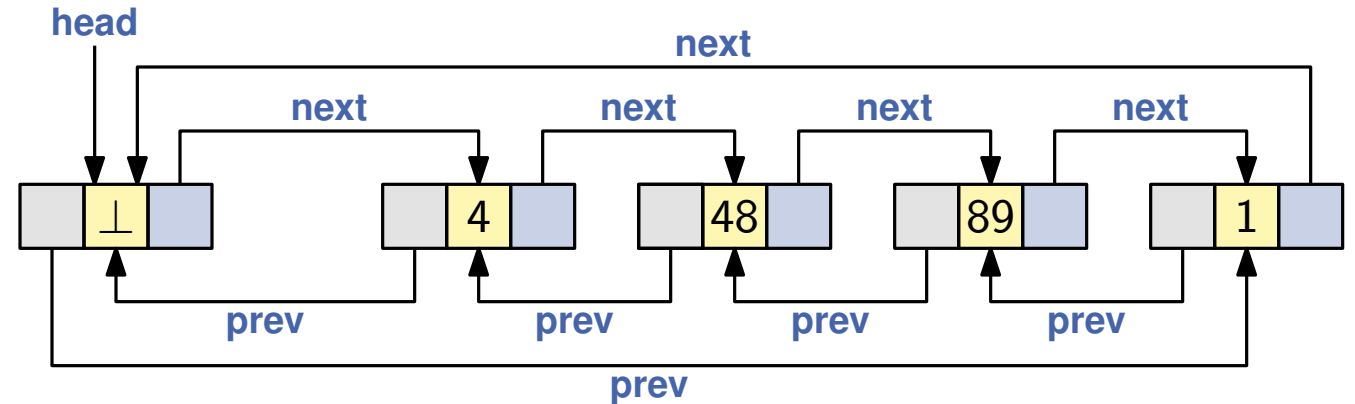
- Gibt es Sonderfälle zu beachten?
- Wie gehe ich mit den  $\perp$ -Pointern um?
- Ziel: Elegante Implementierung, die Sonderfälle reduziert
- Welche nützlichen Operationen gehen sonst noch?



# Implementierungsdetails

## Dummy Knoten $\perp$ für den Kopf

- keine Null-Pointer mehr für erstes und letztes Listenelement
- einfacher, dank Reduktion von Sonderfällen



## insertAfter(Node a, Node x)

// insert node x after node a

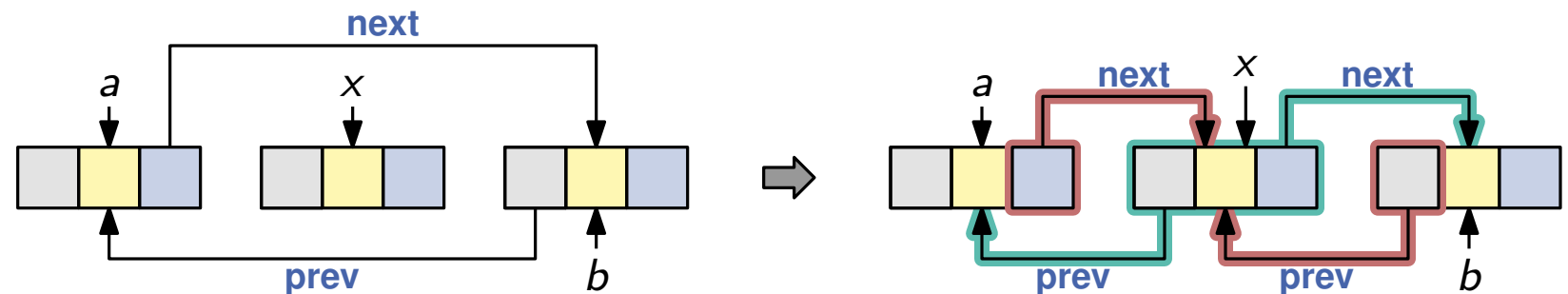
$b := a.next$

$x.prev := a$

$x.next := b$

$a.next := x$

$b.prev := x$

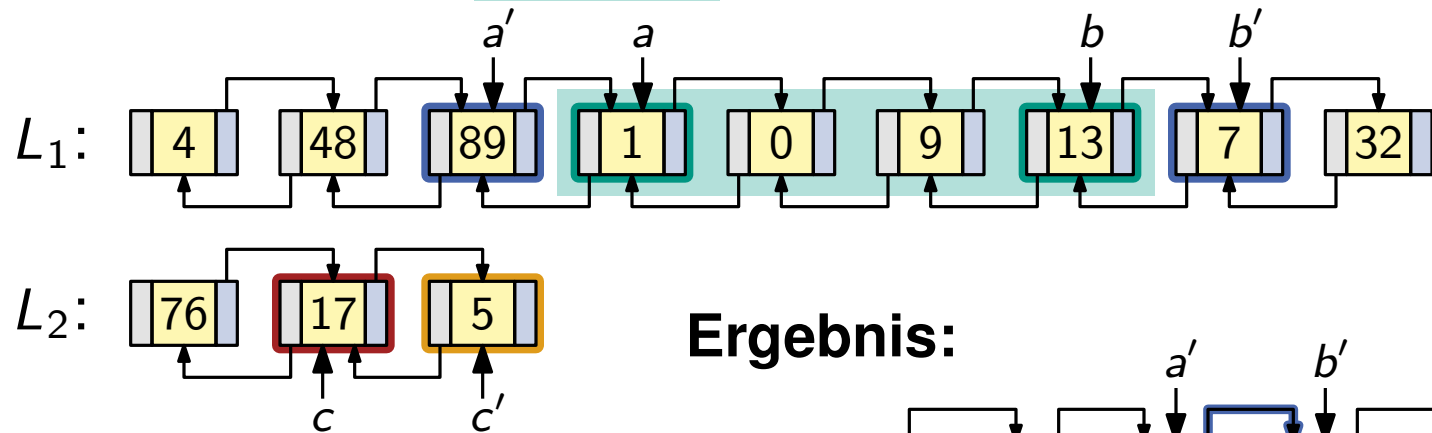


**Beachte:** funktioniert auch dann, wenn  $a = \perp$  oder  $a.next = \perp$

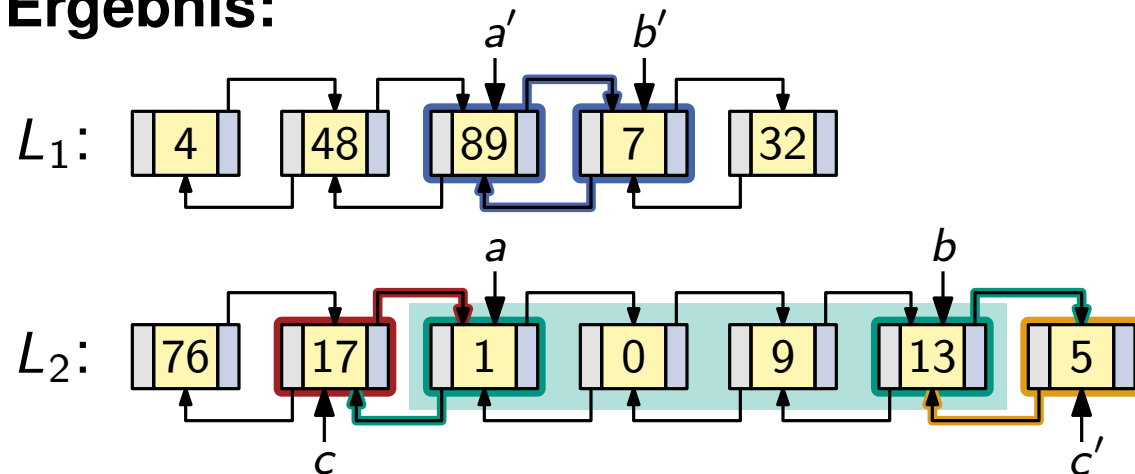
# Mächtigerere Operationen

## Operation: Splice (verbinden, zusammenfügen)

- gegeben: zwei Listen  $L_1$  und  $L_2$  mit Knoten  $a, b \in L_1$  und  $c \in L_2$
- Ziel: verschiebe  $\langle a, \dots, b \rangle$  von  $L_1$  nach  $L_2$  hinter  $c$



### Ergebnis:



Laufzeit:  $\Theta(1)$

**splice**( $a, b, c$ )

// cut out  $\langle a, \dots, b \rangle$

$a' := a.\text{prev}$

$b' := b.\text{next}$

$a'.\text{next} := b'$

$b'.\text{prev} := a'$

// insert after  $c$

$c' := c.\text{next}$

$a.\text{prev} := c$

$b.\text{next} := c'$

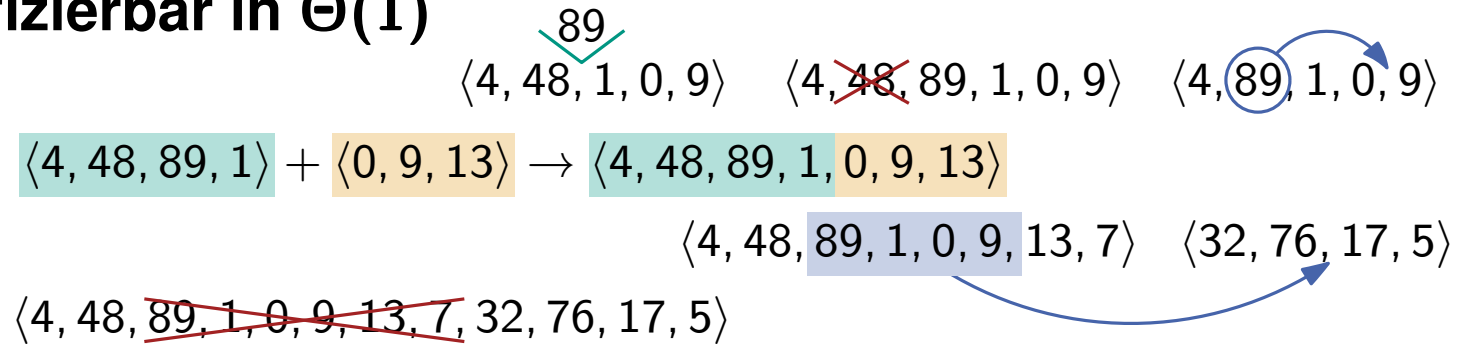
$c.\text{next} := a$

$c'.\text{prev} := b$

# Liste oder Array?

## Stärken der Liste: flexibel modifizierbar in $\Theta(1)$

- Einfügen, Löschen, Verschieben
- Konkatination zweier Listen
- Verschieben ganzer Bereiche
- Löschen ganzer Bereiche



## Schwächen der Liste

- kein wahlfreier Zugriff (**random access**)
- man muss die relevanten Knoten immer schon in der Hand haben
- in der Praxis: schlechtere konstante Faktoren (Speicheroverhead, Cache-Effekte)

### Nebenbemerkung

- $\Theta(1)$  für das Löschen eines Bereichs ist ggf. eine etwas blauäugige Sichtweise
- Was passiert mit dem reservierten Speicher?
- Geben wir den frei? Wer bezahlt das?
- Oder nehmen wir ein memory leak in Kauf?

# Listenvarianten

## Mehr als nur eine Liste

- Liste ist eher ein Konzept als eine einzelne Datenstruktur
- unterschiedliche Anwendungen erfordern im Detail unterschiedliche Implementierungen

## Beispiel 1: Listengröße

- nützliche Information: Größe der Liste (Anzahl Knoten)
- Lösung: speichere diese Info und aktualisiere sie bei Änderungen
- Problem: **splice** wird teurer, weil wir die verschobenen Elemente zählen müssen

## Beispiel 2: Einfach verkettete Liste

- speichere nur **next** aber nicht **prev**
- weniger Speicherplatz, oft schneller
- aber: weniger flexibel, merkwürdige Benutzerschnittstelle (z.B. **removeAfter**)



# Listen in der Wildnis

C++

<https://en.cppreference.com/w/cpp/container/list>

## std::list

std::list is a container that supports constant time insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is usually implemented as a doubly-linked list. Compared to `std::forward_list` this container provides bidirectional iteration capability while being less space efficient.

<b>front</b>	access the first element (public member function)
<b>back</b>	access the last element (public member function)
<b>size</b>	returns the number of elements (public member function)
<b>insert</b>	inserts elements (public member function)
<b>erase</b>	erases elements (public member function)
<b>merge</b>	merges two sorted lists (public member function)
<b>splice</b>	moves elements from another list (public member function)

### std::list<T,Allocator>::splice

Transfers elements from one list to another.

No elements are copied or moved, only the internal pointers of the list nodes are re-pointed.

#### Complexity

1-2) Constant.

3) Constant if other refers to the same object as `*this`, otherwise linear in `std::distance(first, last)`.

### std::list<T,Allocator>::erase

Erases the specified elements from the container.

- 1) Removes the element at pos.
- 2) Removes the elements in the range [first, last).

#### Complexity

- 1) Constant.
- 2) Linear in the distance between first and last.

# Listen in der Wildnis

**Java** <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/LinkedList.html>

## Class `LinkedList<E>`

---

Doubly-linked list implementation of the `List` and `Deque` interfaces. Implements all optional list operations, and permits all elements (including `null`).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

## Interface `ListIterator<E>`

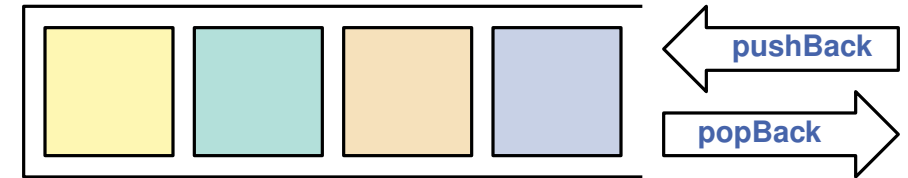
---

Modifier and Type	Method	Description
void	<code>add(E e)</code>	Inserts the specified element into the list (optional operation).
<code>E</code>	<code>next()</code>	Returns the next element in the list and advances the cursor position.
<code>E</code>	<code>previous()</code>	Returns the previous element in the list and moves the cursor position backwards.
void	<code>remove()</code>	Removes from the list the last element that was returned by <code>next()</code> or <code>previous()</code> (optional operation).

# Stapel und Warteschlangen

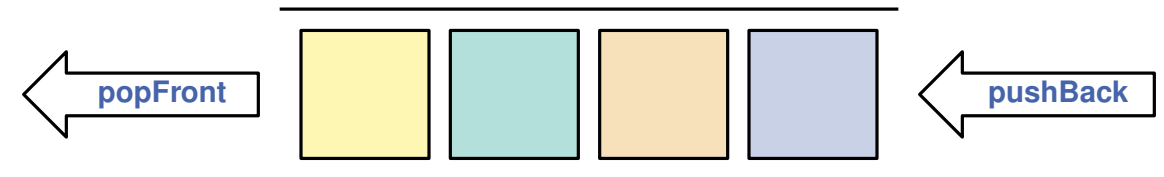
## Stack

- Operationen: **pushBack**, **popBack**
- Implementierung: z.B. mittels Array
- LIFO: Last In – First Out



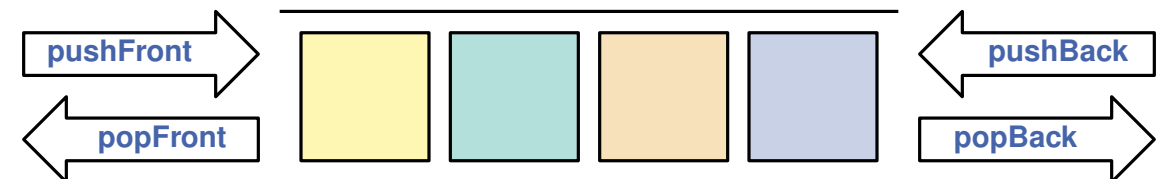
## Queue

- Operationen: **pushBack**, **popFront**
- Implementierung: z.B. mittels Liste
- FIFO: First In – First Out



## Deque – Double-ended Queue

- Operationen: **pushBack**, **popBack**, **pushFront**, **popFront**
- Implementierung: z.B. mittels Liste



# Suchen

## Problemstellung

- gegeben: Folge von  $n$  Zahlen  $A$  (als Array oder Liste) und eine Zahl  $x$
- Ziel: finde  $x$  in der Folge  $A$  (z.B. erstes/jedes Auftreten)

$$45 \stackrel{?}{\in} \langle 4, 48, 89, 1, 0, 9, 13, 7, 32, 76, 17, 5, 37, 28, 82, 63 \rangle$$

## Lineare Suche

- schaue jedes Element aus  $A$  an
- lineare Laufzeit:  $\Theta(n)$

## Geht es besser?

- nur manche Elemente betrachtet  $\rightarrow x$  kann sich unter den nicht angeschauten verstecken
- wir müssen zumindest die Eingabe einmal komplett lesen  $\Rightarrow \Omega(n)$
- also: besser als  $\Theta(n)$  geht nicht, außer wir fordern zusätzliche Eigenschaften für  $A$

# Suchen in sortierten Folgen

## Problemstellung

- gegeben: sortierte Folge von  $n$  Zahlen  $A$  und eine Zahl  $x$
- Ziel: finde  $x$  in der Folge  $A$  (z.B. erstes Auftreten oder Vorgänger)

$45 \stackrel{?}{\in} \langle 0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89 \rangle$



## Binäre Suche

- durch einen Vergleich: entscheide ob  $x$  in der linken oder rechten Hälfte von  $A$  liegt
- suche rekursiv in der relevanten Hälfte von  $A$
- Abbruch: zu durchsuchende Folge ist nur noch konstant groß (hier: 2)

## Anzahl Vergleiche

- pro Vergleich halbiert sich die Größe von  $A \rightarrow$  nur  $\Theta(\log n)$  Halbierungen

## Implementierung

- pro Schritt: Zugriff auf **mittleres** Element im aktuell betrachteten Bereich
- wahlfreier Zugriff  $\rightarrow$  Array

# Abstraktionslevel: Idee → Implementierung

```
binSearchRec(A, x, beg = 0, end = n - 1)
```

```
// find x in ⟨A[beg], ..., A[end]⟩
```

```
if end - beg = 1 then
```

```
  // base case: range has size 2
```

```
  if x = A[beg] then return beg
```

```
  if x = A[end] then return end
```

```
  return between beg and end
```

```
// general case: half the range
```

```
mid := ⌈(beg + end)/2⌉
```

```
if x < A[mid] then
```

```
  return binSearchRec(A, x, beg, mid)
```

```
else
```

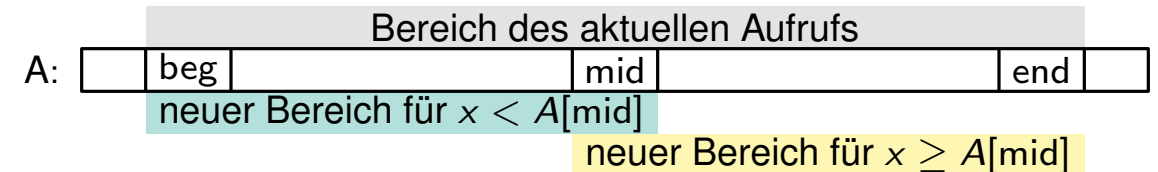
```
  return binSearchRec(A, x, mid, end)
```

## Was müssen wir beweisen?

- ausgegebenes Ergebnis ist **korrekt**
- Terminierung nach **Laufzeit**  $\Theta(\log n)$

## Korrektheitsbeweis mittels Invarianten

- zeige, dass wir immer im richtigen Teilbereich suchen:  $A[beg] \leq x \leq A[end]$



- wenn  $A[beg] \leq x \leq A[end]$  im aktuellen Aufruf gilt
- dann auch im nächsten

### Achtung am Anfang:

$A[0] \leq x \leq A[n - 1]$  muss nicht gelten!

# Implementierung: zweiter Versuch

## Was genau wollen wir haben?

- Index  $i$ , sodass:  $A[i] = x$  (falls  $x \in A$ )  
 oder  $A[i - 1] < x < A[i]$  (falls  $x \notin A$ )  
(Konvention:  $A[-1] = -\infty, A[n] = \infty$ )
- Invariante für diesen Index  $i$ :  $i \in [\text{beg}, \text{end}]$

## Beweis der Invariante

- gilt am Anfang mit  $\text{beg} = 0$  und  $\text{end} = n$
- für den Erhalt der Invariante, prüfe 4 Fälle:
  - Fall 1.1:  $x \in A$  und  $x \leq A[\text{mid}]$
  - Fall 1.2:  $x \in A$  und  $x > A[\text{mid}]$
  - Fall 2.1:  $x \notin A$  und  $x \leq A[\text{mid}]$
  - Fall 2.2:  $x \notin A$  und  $x > A[\text{mid}]$

**Basisfall:** mit der Invariante sehr einfach

```
binSearchRec(A, x, beg = 0, end = n)
```

```
// find  $i \in [\text{beg}, \text{end}]$  with this property
```

```
if beg = end then return beg
```

```
// general case: half the range
```

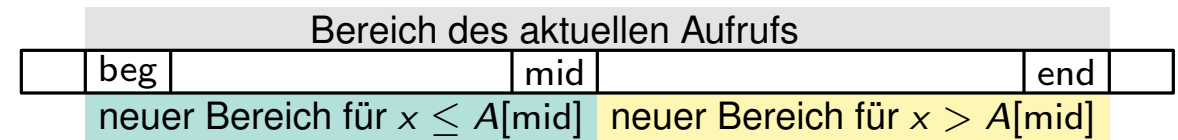
```
mid :=  $\lfloor (\text{beg} + \text{end}) / 2 \rfloor$ 
```

```
if  $x \leq A[\text{mid}]$  then
```

```
  return binSearchRec(A, x, beg, mid)
```

```
else
```

```
  return binSearchRec(A, x, mid + 1, end)
```



# Implementierung: zweiter Versuch

## Was genau wollen wir haben?

- Index  $i$ , sodass:  $A[i] = x$  (falls  $x \in A$ )  
 oder  $A[i-1] < x < A[i]$  (falls  $x \notin A$ )  
(Konvention:  $A[-1] = -\infty, A[n] = \infty$ )
- Invariante für diesen Index  $i$ :  $i \in [\text{beg}, \text{end}]$

## Beweis der Laufzeit

- $\text{end} - \text{beg}$  wird mindestens halbiert:

$$\begin{aligned} \text{mid} - \text{beg} &= \left\lfloor \frac{\text{beg} + \text{end}}{2} \right\rfloor - \text{beg} \\ &\leq \frac{\text{beg} + \text{end}}{2} - \text{beg} = \frac{\text{end} - \text{beg}}{2} \end{aligned}$$

$$\begin{aligned} \text{end} - (\text{mid} + 1) &= \text{end} - \left( \left\lfloor \frac{\text{beg} + \text{end}}{2} \right\rfloor + 1 \right) \\ &\leq \text{end} - \frac{\text{beg} + \text{end}}{2} \\ &= \frac{\text{end} - \text{beg}}{2} \end{aligned}$$

**binSearchRec**( $A, x, \text{beg} = 0, \text{end} = n$ )

// find  $i \in [\text{beg}, \text{end}]$  with this property

**if**  $\text{beg} = \text{end}$  **then return**  $\text{beg}$

// general case: half the range

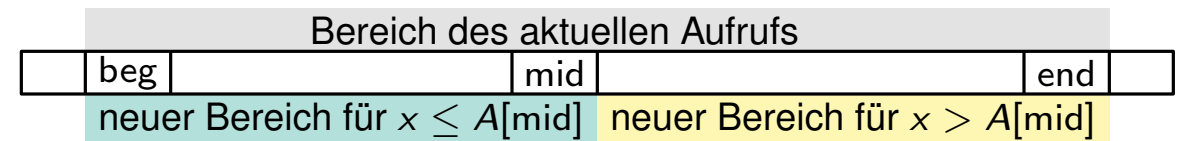
$\text{mid} := \lfloor (\text{beg} + \text{end}) / 2 \rfloor$

**if**  $x \leq A[\text{mid}]$  **then**

**return** **binSearchRec**( $A, x, \text{beg}, \text{mid}$ )

**else**

**return** **binSearchRec**( $A, x, \text{mid} + 1, \text{end}$ )





# Binäre Suche

## Theorem

Sei  $M$  eine geordnete Menge (z.B.  $\mathbb{Z}$ ). Sei  $A$  ein sortiertes Array der Länge  $n$  mit Werten aus  $M$  und sei  $x \in M$ . Die **binäre Suche** findet den Index  $i$  mit  $A[i - 1] < x \leq A[i]$  in  $\Theta(\log n)$  Vergleichen.

(Konvention:  $A[-1] = -\infty$ ,  $A[n] = \infty$ )

## Folgerungen: Bereichsanfragen

- für  $a, b \in M$  können wir in  $\Theta(\log n)$  herausfinden, wie viele Elemente  $A$  aus  $[a, b]$  enthält
- $A$  enthält  $k$  Elemente aus  $[a, b] \rightarrow$  wir können sie in  $\Theta(\log n + k)$  aufzählen

## Einfache Idee, mit Stolperfallen in der Umsetzung

- abstrakte Idee: vergleiche in jedem Schritt mit mittlerem Element  $\rightarrow$  halbiere Suchraum
- Umsetzung: man muss mit Randfällen etwas aufpassen
- hilfreiche Technik: Korrektheitsbeweis mittels Invariante

# Schneller geht es nicht

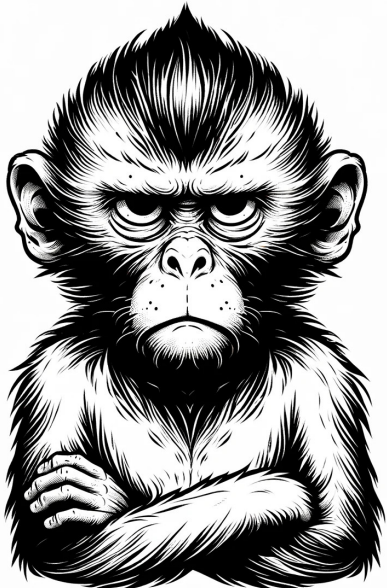
## Theorem

Man kann nicht in  $o(\log n)$  suchen.

**Sehr verkürzte Darstellung!**

## Theorem

Es gibt keine Datenstruktur die für jede geordnete Menge  $M$  und jede Teilmenge  $M' \subseteq M$  mit  $n := |M'|$  berechnet werden kann, die für jedes  $x \in M$  in  $o(\log n)$  Zeit testet, ob  $x \in M'$ .



**Warum ist das so umständlich?**

**Sagt die einfachere Formulierung nicht das gleiche!?!**

- hier wichtig: wir wissen von  $M$  nur, dass es eine geordnete Menge ist
- das einzige was wir mit Elementen aus  $M$  tun können: Ordnungsrelation überprüfen  $\rightarrow$  jede Entscheidung basiert auf einem Vergleich
- wir sagen auch: **vergleichsbasiertes** Suchen geht nicht in  $o(\log n)$
- für manche Mengen  $M$  kann man tatsächlich in  $o(\log n)$  suchen

(später auf Übungsblatt)

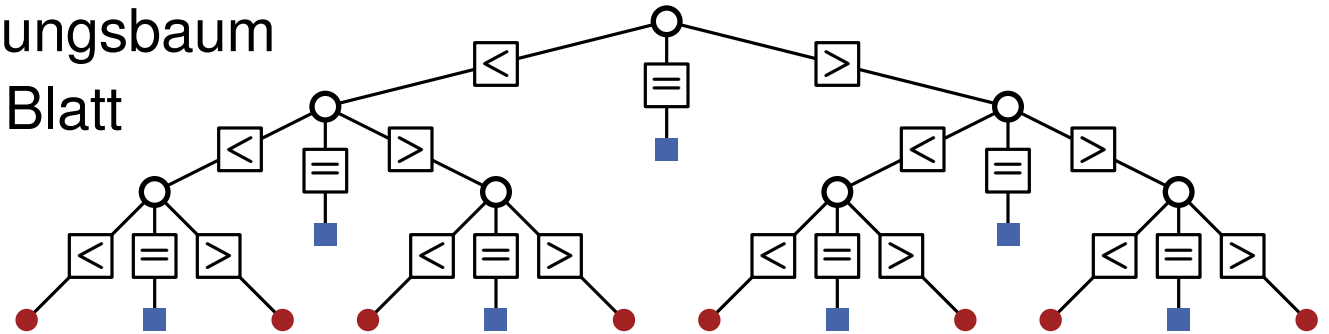
# Schneller geht es nicht

## Theorem

Es gibt keine Datenstruktur die für jede geordnete Menge  $M$  und jede Teilmenge  $M' \subseteq M$  mit  $n := |M'|$  berechnet werden kann, die für jedes  $x \in M$  in  $o(\log n)$  Zeit testet, ob  $x \in M'$ .

## Beweis

- Ausführung der Suche hängt nur von Vergleichen zwischen  $x$  und Elementen in  $M'$  ab
- Menge aller Ausführungen: Entscheidungsbaum
- eine Ausführung: Pfad von Wurzel zu Blatt  
→ Pfadlänge = Anzahl Vergleiche
- bei „=“ wird terminiert, da  $x$  gefunden  
→ blaues Blatt ■
- jedes  $x \in M'$  findet man an einem andern blauen Blatt
- höchstens  $k$  Vergleiche  $\Rightarrow$  höchstens  $\sum_{i=0}^{k-1} 2^i = 2^k - 1$  verschiedene blaue Blätter
- für  $k < \log_2 n$  kann nicht für jedes  $x \in M'$  das richtige Ergebnis herauskommen



# Zusammenfassung

## Listen

- verzeigerte Datenstruktur zur Speicherung einer Folge
- flexibel modifizierbar: z.B. schnelles Einfügen, Löschen, Splice
- kein wahlfreier Zugriff mittels Index (kein random access)

## Binäre Suche

- |  | <b>Abstraktionsebene</b>                                    |
|--|---|
| ■ Voraussetzung: sortierte Folge und wir haben random access   |   |
| ■ Vergleich mit mittlerem Element des aktuellen Suchraums<br>→ halbiert Suchraum → $\Theta(\log n)$ Vergleiche | <b>hoch</b><br>(Ideenfindung, Abschätzung der Laufzeit)     |
| ■ Korrektheit der Detailumsetzung via Invariante   | <b>niedrig</b><br>(Formalisierung, Weg zur Implementierung) |

## Untere Schranke

- besser als  $\Theta(\log n)$  geht es nicht
- außer, wenn wir Annahmen darüber machen, auf welcher Art von Daten wir suchen