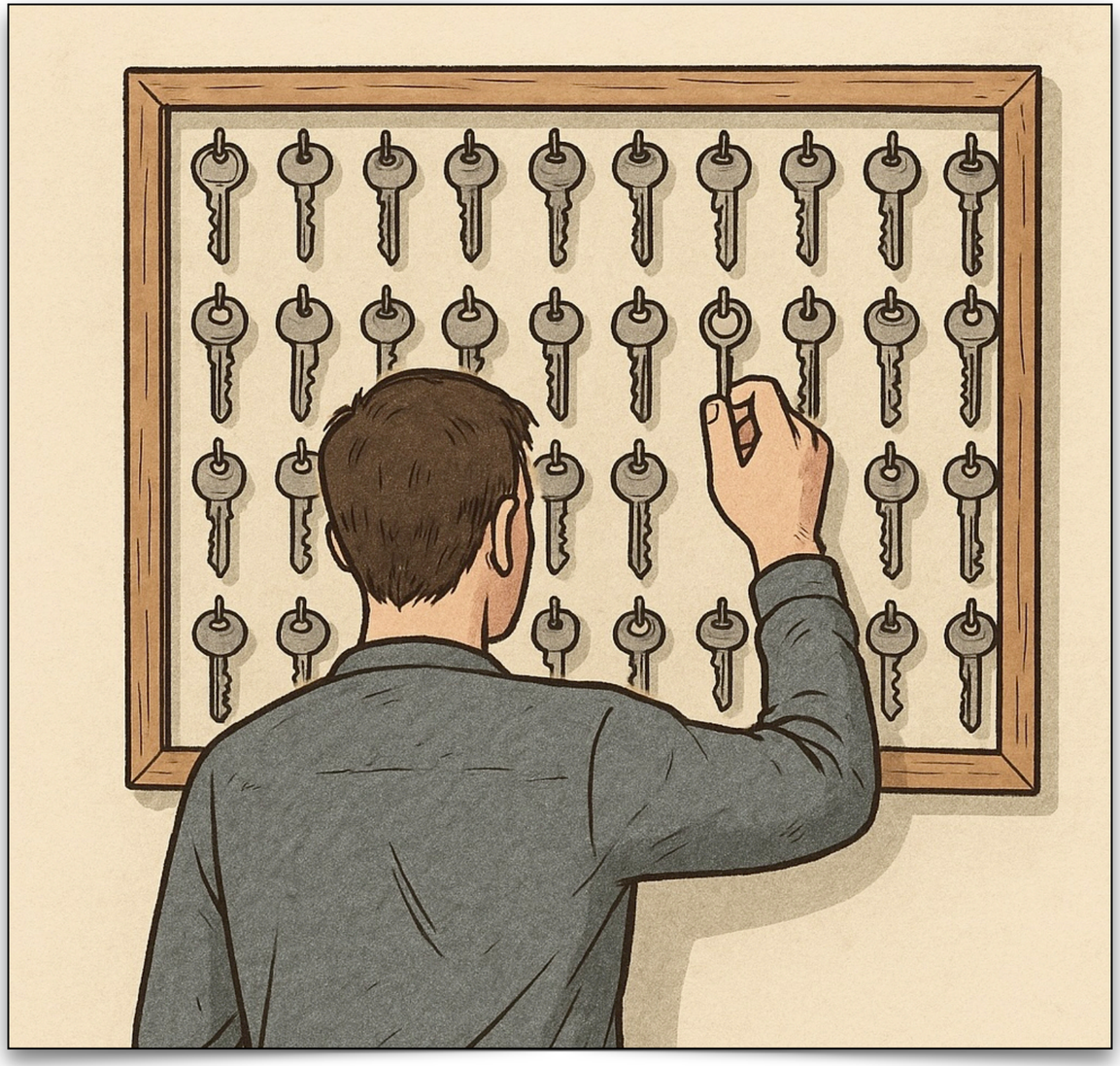
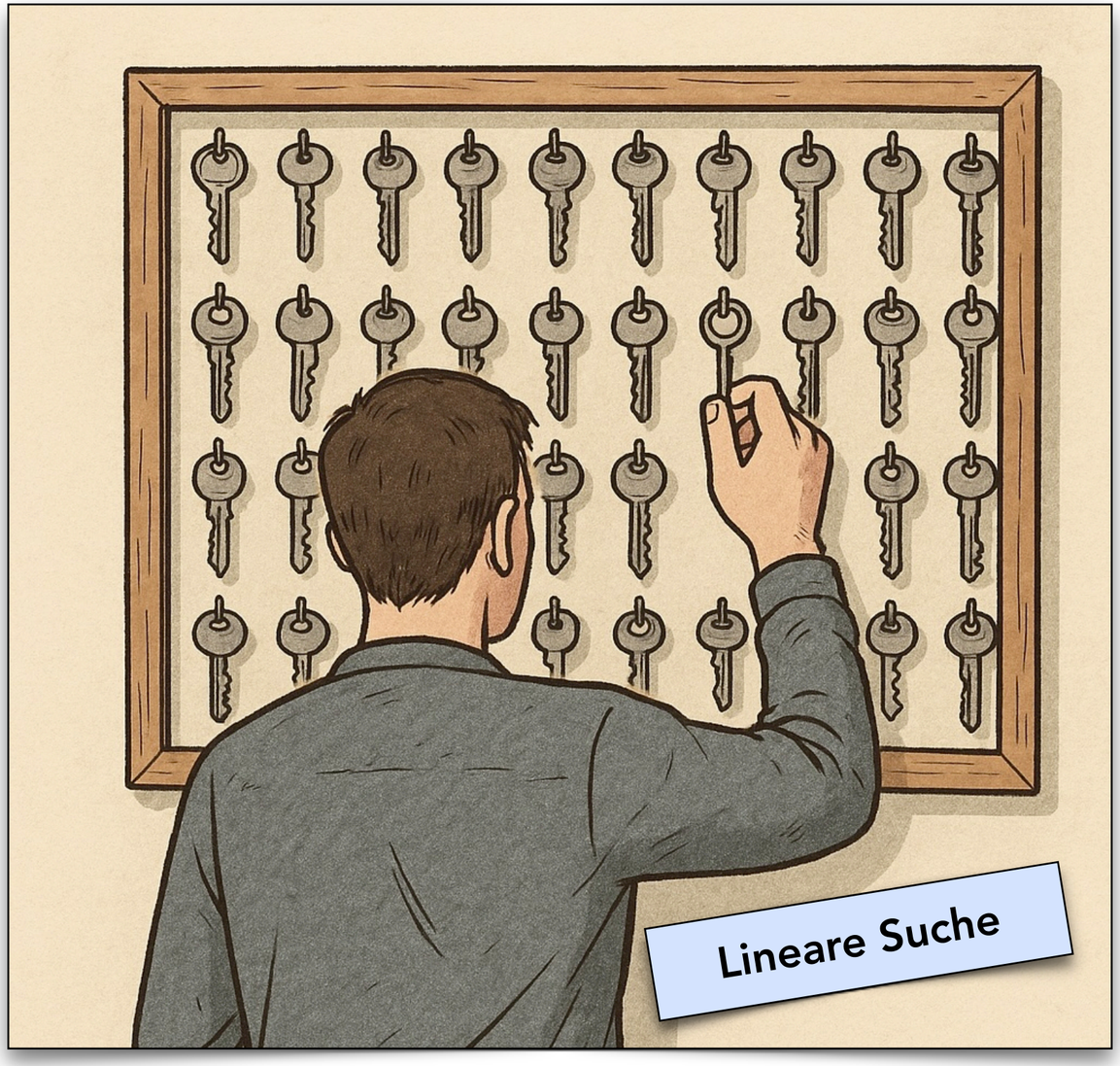
Folge 6: Suchverfahren

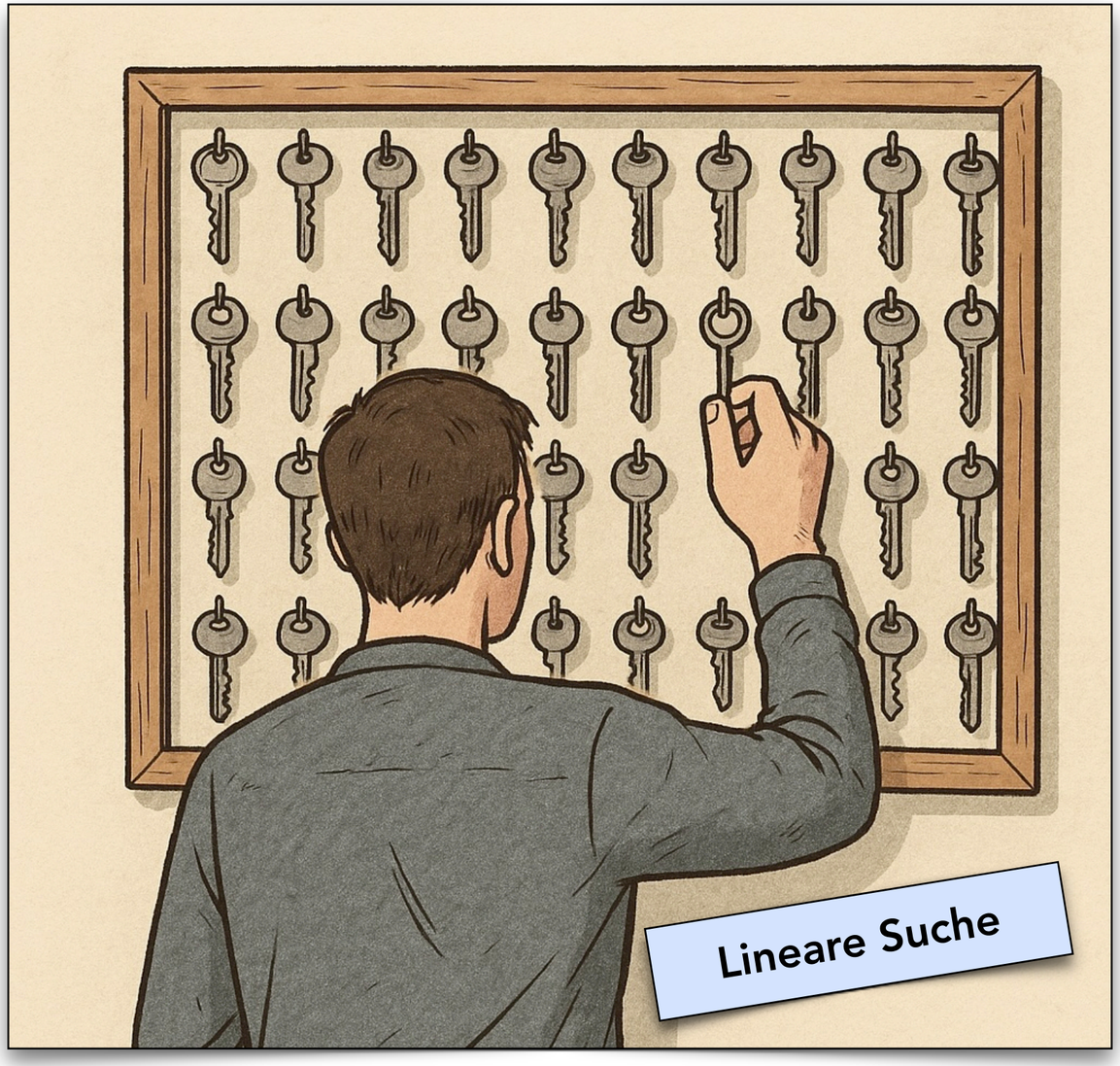
6. Suchverfahren



6. Suchverfahren



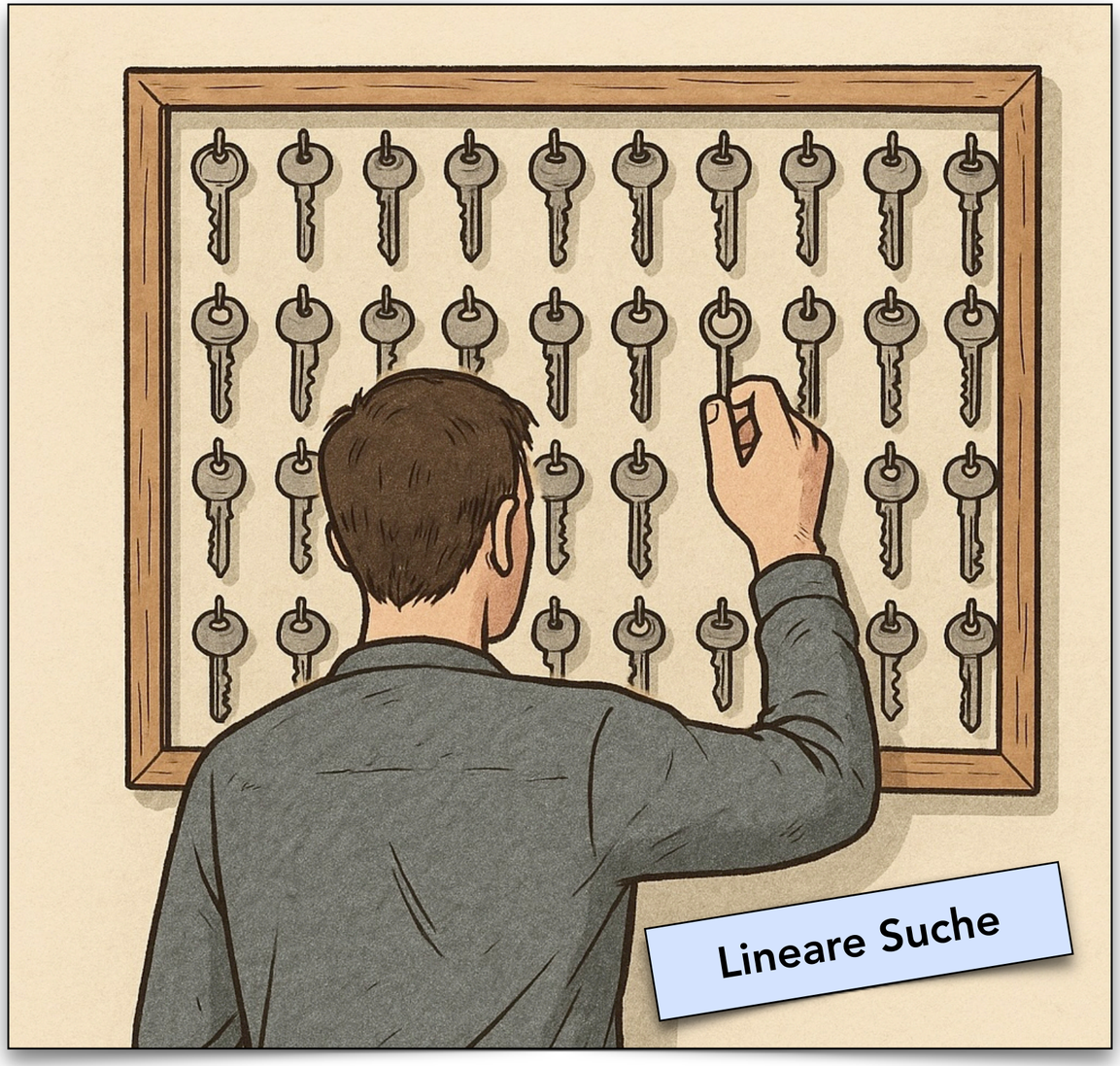
6.1 Die lineare Suche: Definition



Definition / Merksatz:

Bei der ***linearen*** oder ***sequentiellen*** Suche wird jedes Element einer ungeordneten Liste nacheinander überprüft, bis entweder das gesuchte Element gefunden wurde oder alle Einträge durchsucht sind.

6.1 Die lineare Suche: Zeitverhalten



Best case s = 1

Element direkt am Anfang der Liste.

Worst case s = n

Element am Ende der Liste.
Element nicht in der Liste enthalten.

Average case s = (n+1)/2

Element irgendwo in der Liste enthalten.

Merke:

Zeitkomplexität bzw. ***Zeitverhalten*** der linearen Suche ist O(N).

6.1 Die lineare Suche: Java-Methoden

```
public int getPos(int[] array, int suchzahl)
{
   for (int pos=0; pos < array.length; pos++)
   {
      if (suchzahl == array[pos])
         return pos;
   }
   return -1;
}
```

```
public boolean contains(int[] array, int suchzahl)
{
   return getPos(array, suchzahl) != -1;
}
```

6.1 Die lineare Suche: Suche in Strings

Suchmethoden der Klasse String
1. Die indexOf(...) - Methoden

Diese vier Methoden suchen den Index des ersten Vorkommens des Zeichens `ch` oder des Strings `s`, gegebenenfalls aber einer Startposition `form`.

`int indexOf(int ch)`
Sucht den Index des ersten Vorkommen des Zeichen `ch`.

`int indexOf(int ch, int from)`
Sucht das Zeichen `ch` ab der Startposition `from`.

`int indexOf(String s)`
Sucht das erste Vorkommen des Strings `s`.

`int indexOf(String s, int from)`
Sucht `s` ab der Startposition `from`.

6.1 Die lineare Suche: Suche in Strings

Suchmethoden der Klasse String
2. Die contains(...) und starts/endsWith(...) - Methoden

Diese drei Methoden überprüfen, ob ein Substring **s** in dem String-Objekt vorkommt.

`boolean contains(String s)`
Liefert true zurück, falls der String s im String-Objekt enthalten ist.

`boolean startsWith(String s)`
Liefert true zurück, falls das String-Objekt mit dem Präfix s beginnt.

`boolean endsWith(String s)`
Liefert true zurück, falls das String-Objekt mit dem Suffix s abschließt.

6.1 Die lineare Suche: Suche in Strings

Suchmethoden der Klasse String
3. die charAt(...) - Methode

Diese Methode liefert das Zeichen an einer bestimmten Position des Strings. Wird gern in for-Schleifen verwendet.

`char charAt(int index)`
Liefert das Zeichen an der Position **`index`**.

Definition / Merksatz:

Die Suchmethoden der Klasse **String** durchsuchen die Zeichenfolge ***linear***.
Daher ist der Zeitaufwand O(n).

6.2 Die binäre Suche

6.2 Die binäre Suche: Beispiel

Ein sortierter Array aus 30 int-Zahlen

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

6.2 Die binäre Suche: Beispiel

Ein sortierter Array aus 30 int-Zahlen. Gesucht wird nach der **16**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

Gedankliches Aufteilen des Arrays in zwei Hälften. Die 16 müsste in der linken Hälfte vorkommen.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

6.2 Die binäre Suche: Beispiel

Ein sortierter Array aus 30 int-Zahlen. Gesucht wird nach der **16**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

Gedankliches Aufteilen des Arrays in zwei Hälften. Die 16 müsste in der linken Hälfte vorkommen.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

Gedankliches Aufteilen der linken Hälfe in zwei Hälften. Die 16 müsste in der rechten Hälfte vorkommen.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

6.2 Die binäre Suche: Beispiel

Gedankliches Aufteilen der rechten Hälfte in zwei Hälften. Die 16 müsste in der linken Hälfte vorkommen.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

Gedankliches Aufteilen der linken Hälfte in zwei Hälften. Die 16 müsste in der linken Hälfte vorkommen.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

Gedankliches Aufteilen der linken Hälfte in zwei Hälften. Beide Hälften bestehen nur noch aus einem Element.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

Die 16 kommt weder in dem linken noch in dem rechten Arrayelement vor. Also ist sie nicht im Array enthalten.

6.2 Die binäre Suche: Java-Quelltext

```
public int sucheBinaer(int[] a, int suchzahl)
{
    int links    =  0;
    int rechts   =  a.length-1;

    while (links <= rechts)
    {
        int mitte = (links + rechts) / 2;

        if (a[mitte] == suchzahl)
            return a;
        else if (a[mitte] > suchzahl)
            rechts = mitte-1;
        else
            links = mitte+1;
    }
    return -1; // suchzahl nicht gefunden
}
```

Berechnung der Mitte des Arrays / Teilarrays

6.2 Die binäre Suche: Java-Quelltext

```
public int sucheBinaer(int[] a, int suchzahl)
{
    int links    =  0;
    int rechts   =  a.length-1;

    while (links <= rechts)
    {
        int mitte = (links + rechts) / 2;

        if (a[mitte] == suchzahl)
            return a;
        else if (a[mitte] > suchzahl)
            rechts = mitte-1;
        else
            links = mitte+1;
    }
    return -1; // suchzahl nicht gefunden
}
```

Zahl in der Mitte gefunden.

6.2 Die binäre Suche: Java-Quelltext


```
public int sucheBinaer(int[] a, int suchzahl)
{
    int links    =  0;
    int rechts   =  a.length-1;

    while (links <= rechts)
    {
        int mitte = (links + rechts) / 2;

        if (a[mitte] == suchzahl)
            return a;
        else if (a[mitte] > suchzahl)
            rechts = mitte-1;
        else
            links = mitte+1;
    }
    return -1; // suchzahl nicht gefunden
}
```

Links oder rechts der Mitte weitersuchen ?

6.2 Die binäre Suche: Java-Quelltext

```
public int sucheBinaer(int[] a, int suchzahl)
{
    int links    =  0;
    int rechts   =  a.length-1;

    while (links <= rechts)
    {
        int mitte = (links + rechts) / 2;

        if (a[mitte] == suchzahl)
            return a;
        else if (a[mitte] > suchzahl)
            rechts = mitte-1;
        else
            links = mitte+1;
    }
    return -1; // suchzahl nicht gefunden
}
```

Links oder rechts der Mitte weitersuchen ?

Ermittlung des neuen (gedanklichen) Teilarrays durch Neuberechnung von rechts bzw. links.

6.2 Die binäre Suche: Zeitverhalten

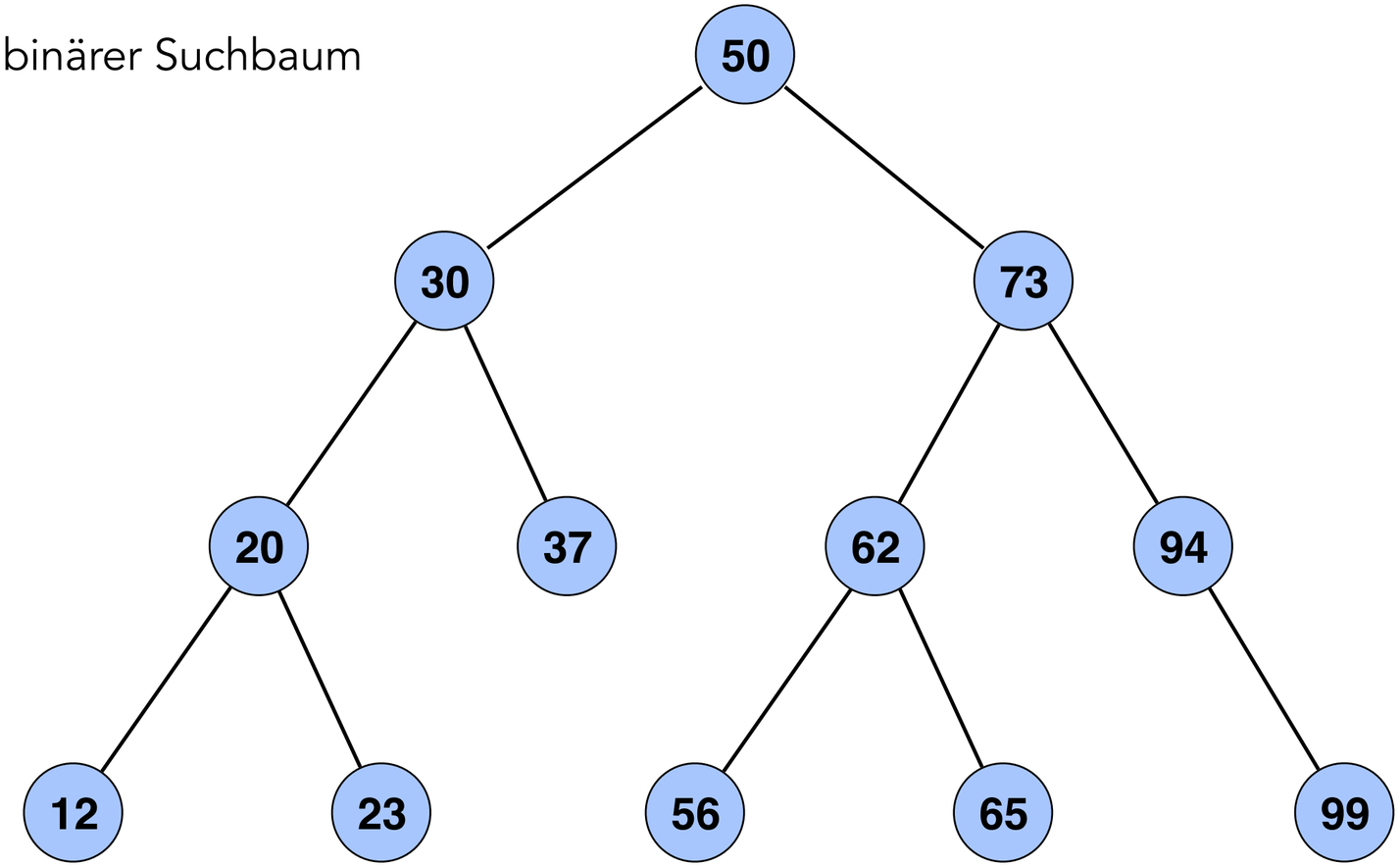
Anzahl der Zahlen N	maximale Vergleiche
1	1
2	2
4	3
8	4
16	5
32	6
64	7
128	8
256	9
512	10

Anzahl der Zahlen N	maximale Vergleiche
1.024	11
2.048	12
4.096	13
8.192	14
16.384	15
32.768	16
65.536	17
131.072	18
262.144	19
524.288	20

O(log N)

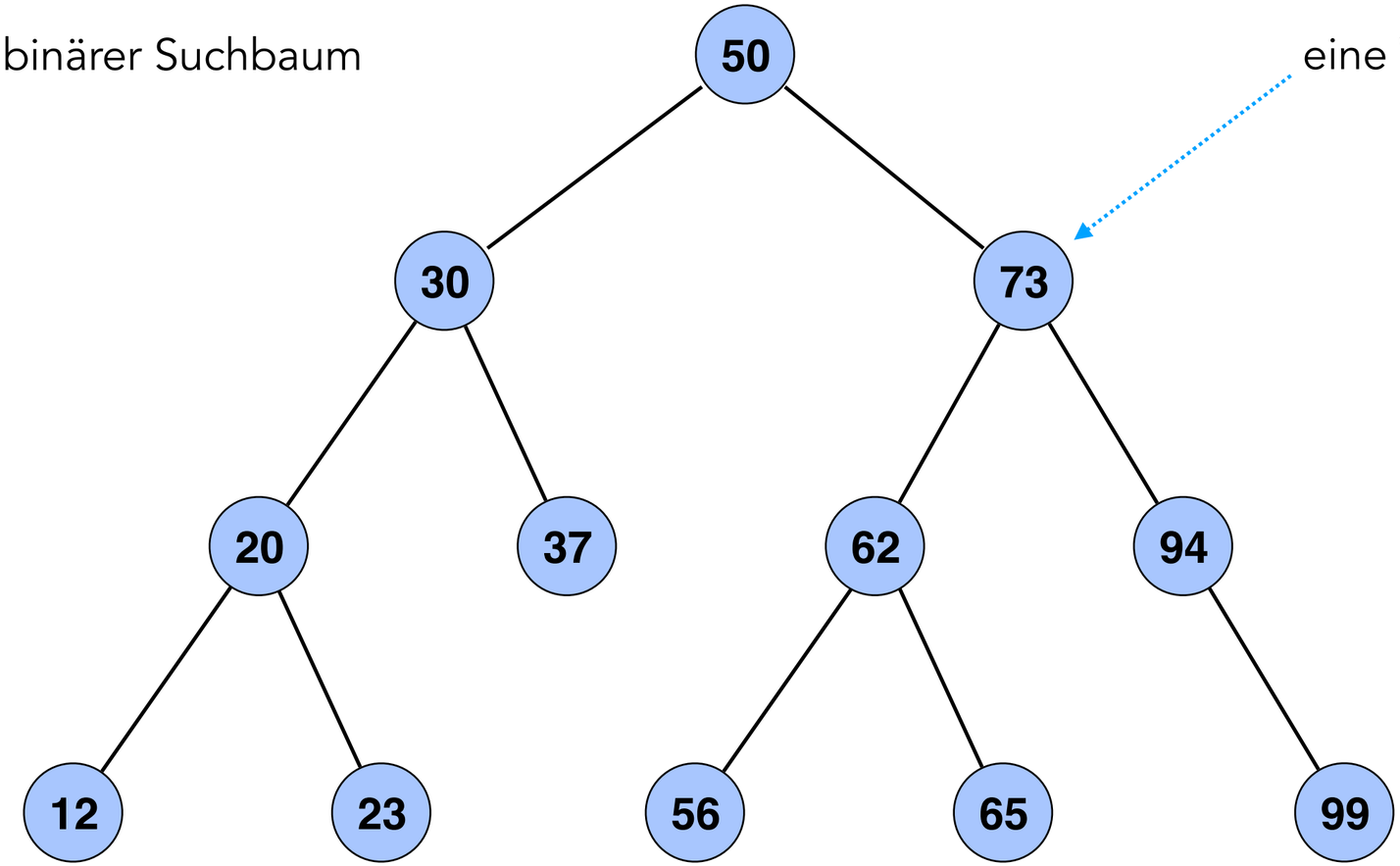
6.2 Die binäre Suche: Exkurs binäre Suchbäume

Ein binärer Suchbaum



6.2 Die binäre Suche: Exkurs binäre Suchbäume

Ein binärer Suchbaum



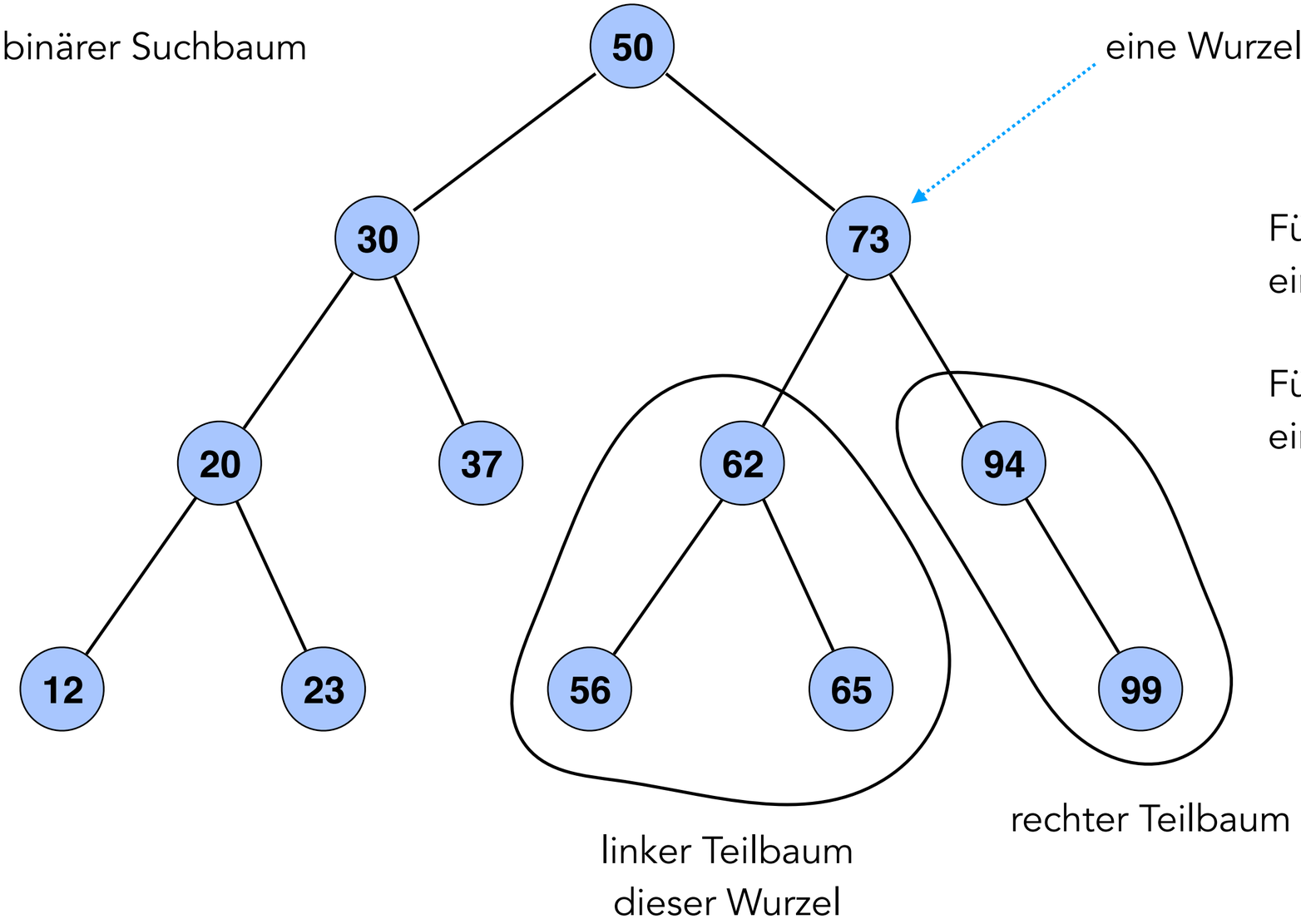
eine Wurzel

Für alle Elemente E im linken Teilbaum einer Wurzel W gilt: **E < W**

Für alle Elemente E im rechten Teilbaum einer Wurzel W gilt: **E >= W**

6.2 Die binäre Suche: Exkurs binäre Suchbäume

Ein binärer Suchbaum



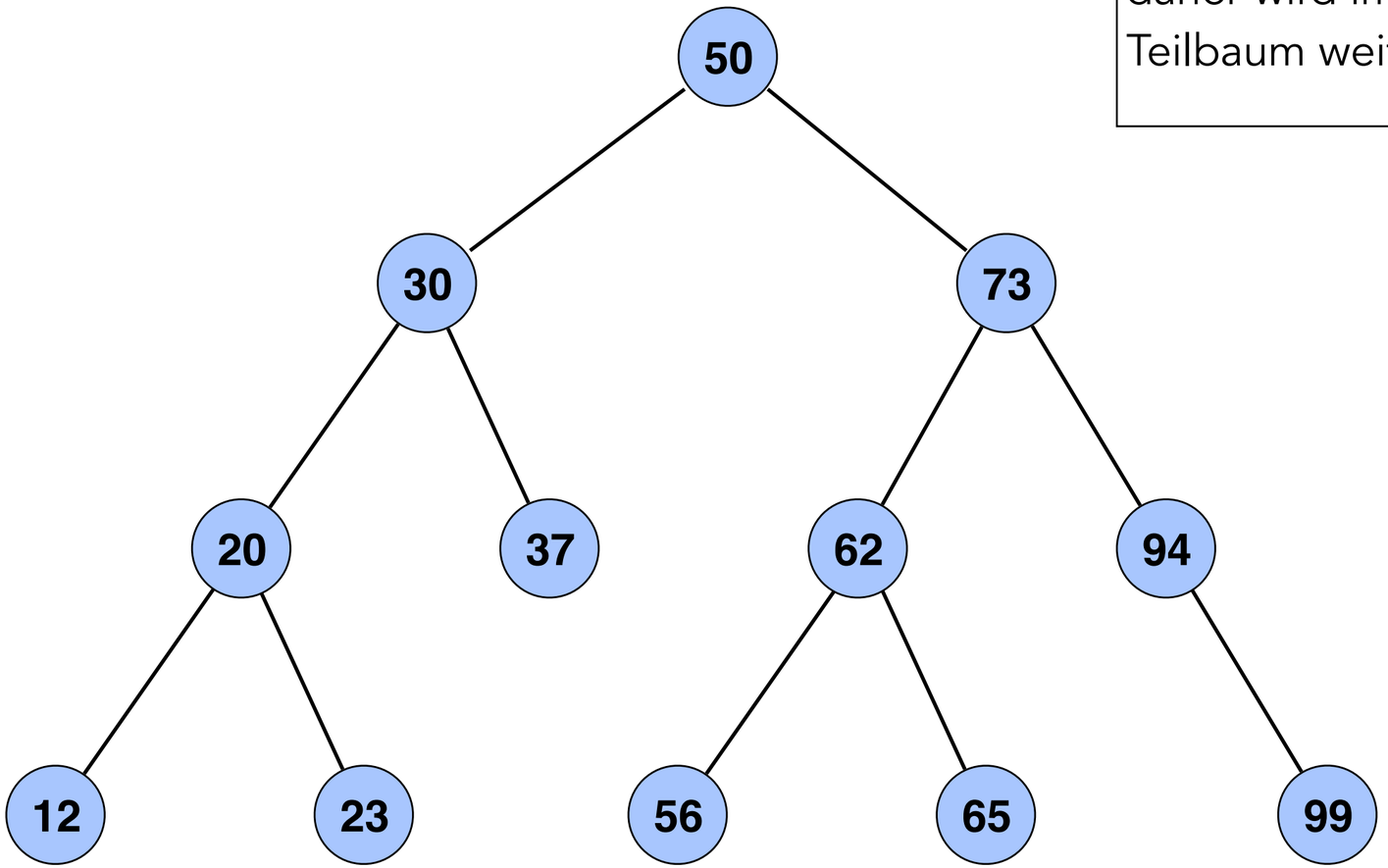
Für alle Elemente E im linken Teilbaum einer Wurzel W gilt: **E < W**

Für alle Elemente E im rechten Teilbaum einer Wurzel W gilt: **E >= W**

6.2 Die binäre Suche: Exkurs binäre Suchbäume

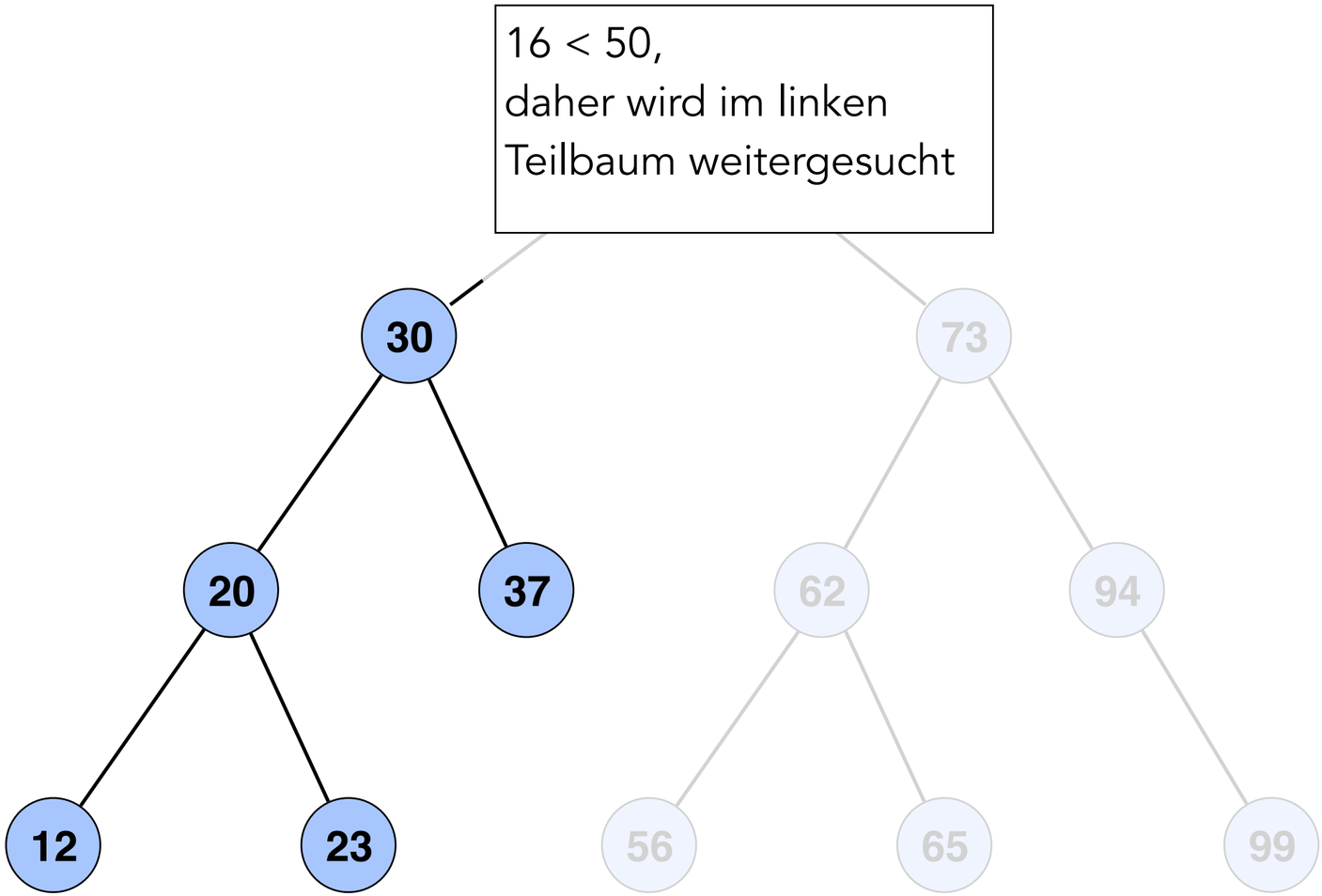
Suchen nach der 16

16 < 50,
daher wird im linken
Teilbaum weitergesucht



6.2 Die binäre Suche: Exkurs binäre Suchbäume

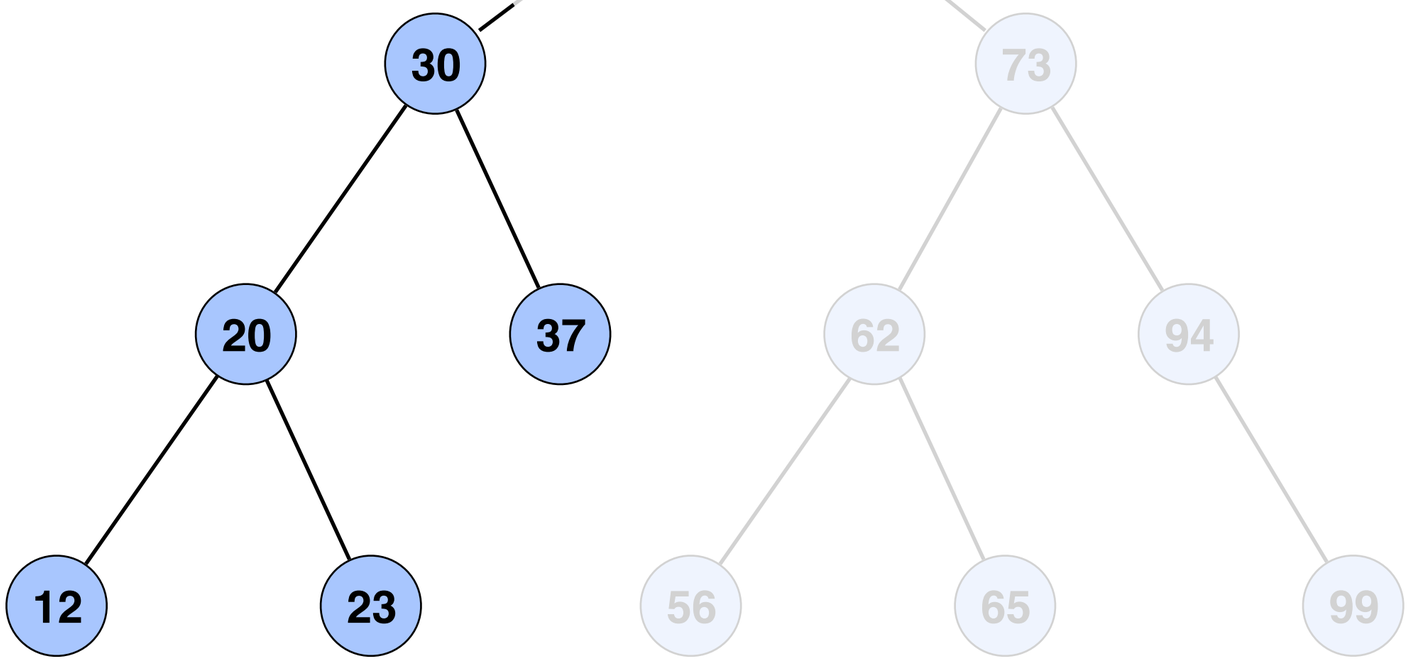
Suchen nach der 16



6.2 Die binäre Suche: Exkurs binäre Suchbäume

Suchen nach der 16

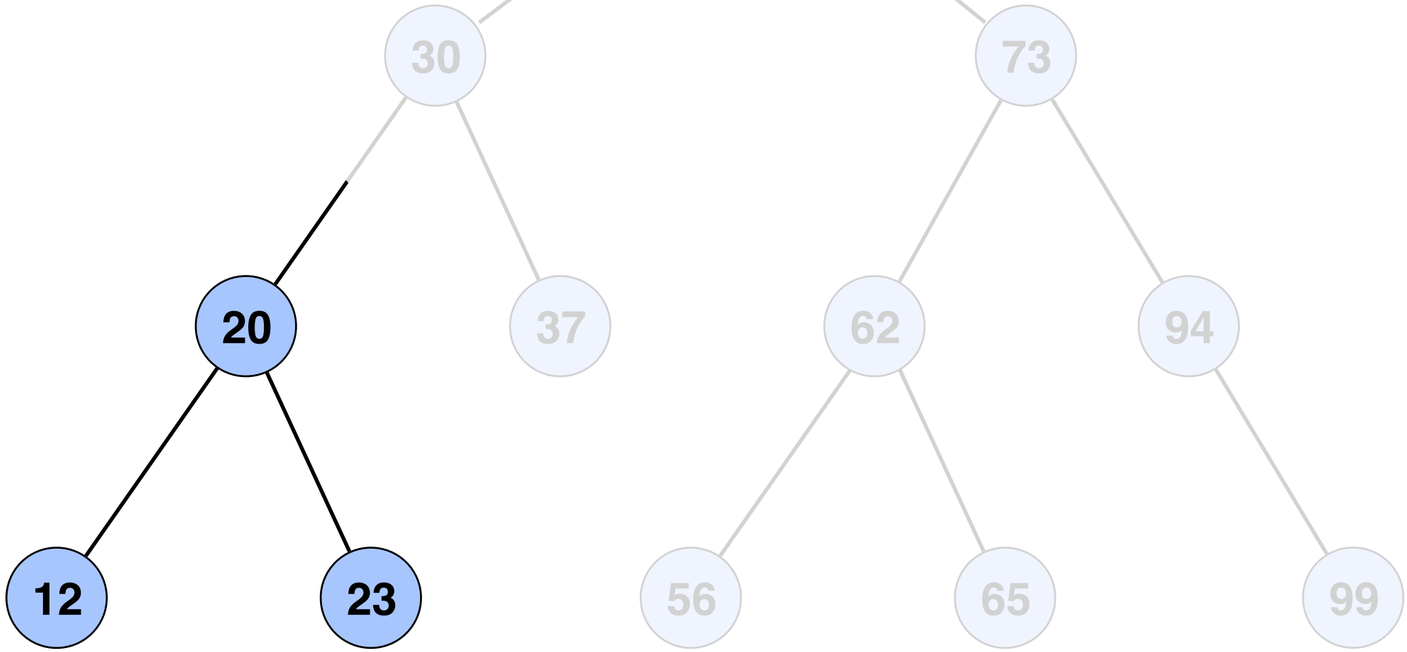
16 < 30,
daher wird im linken
Teilbaum weitergesucht



6.2 Die binäre Suche: Exkurs binäre Suchbäume

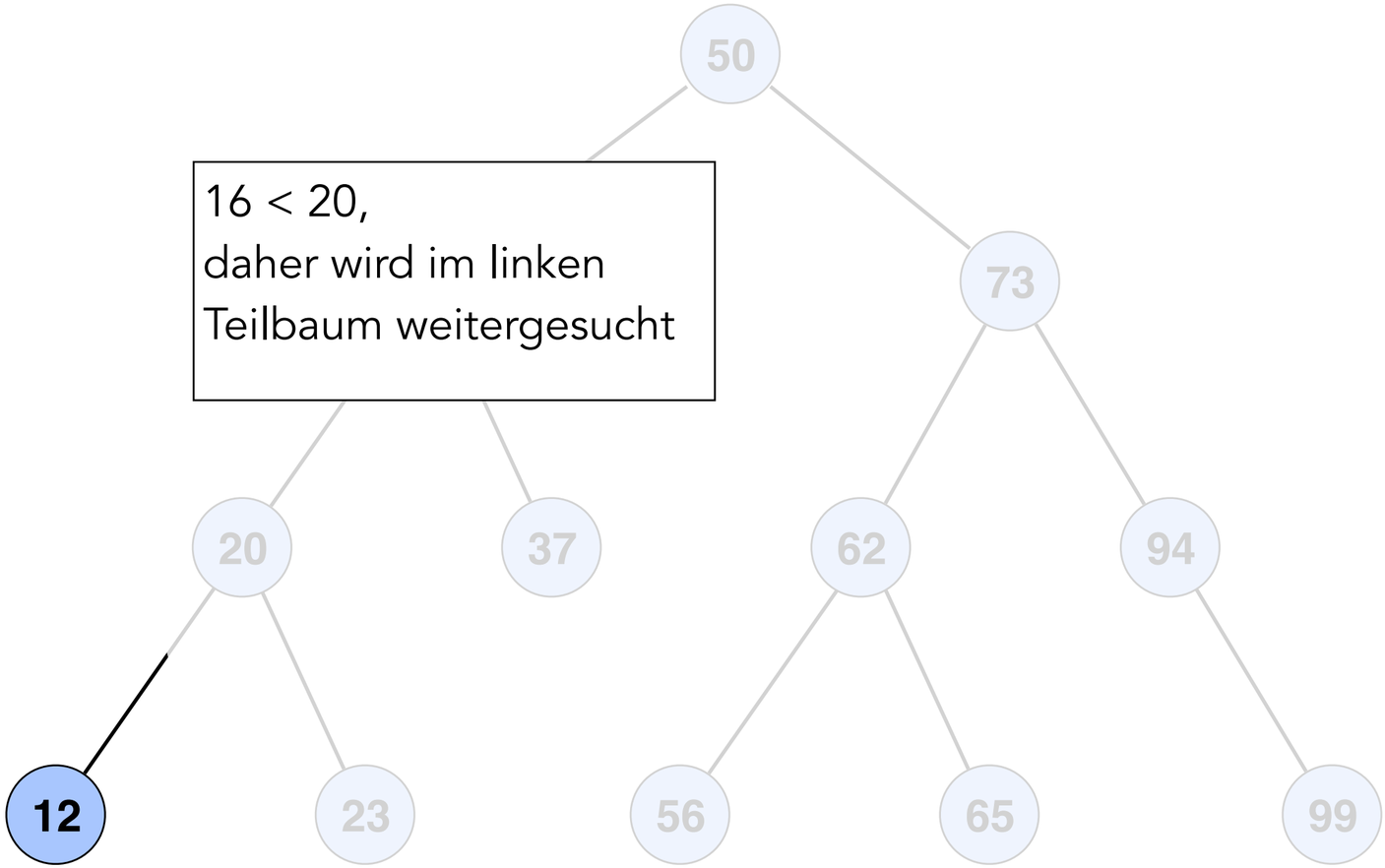
Suchen nach der 16

16 < 30,
daher wird im linken
Teilbaum weitergesucht



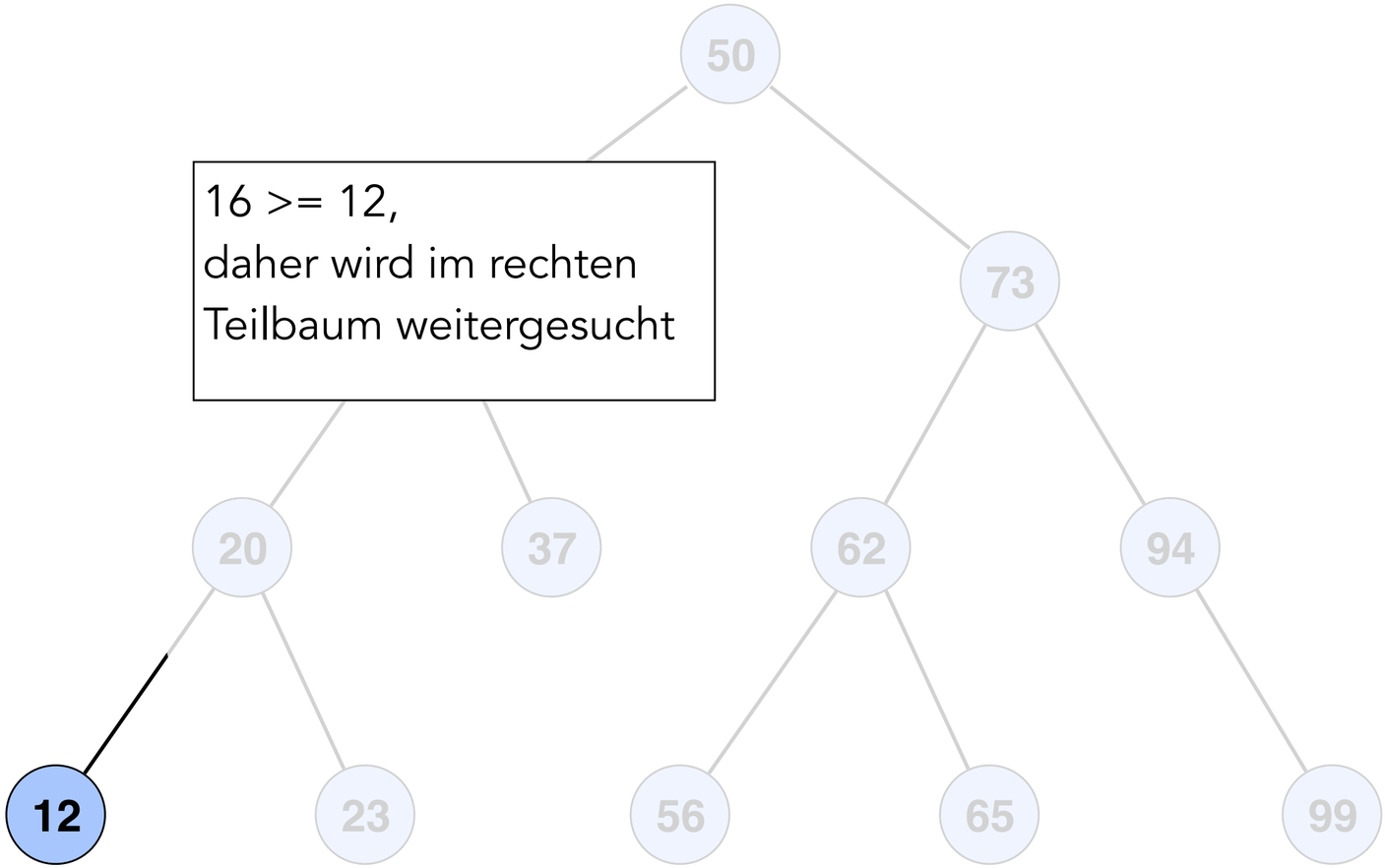
6.2 Die binäre Suche: Exkurs binäre Suchbäume

Suchen nach der 16



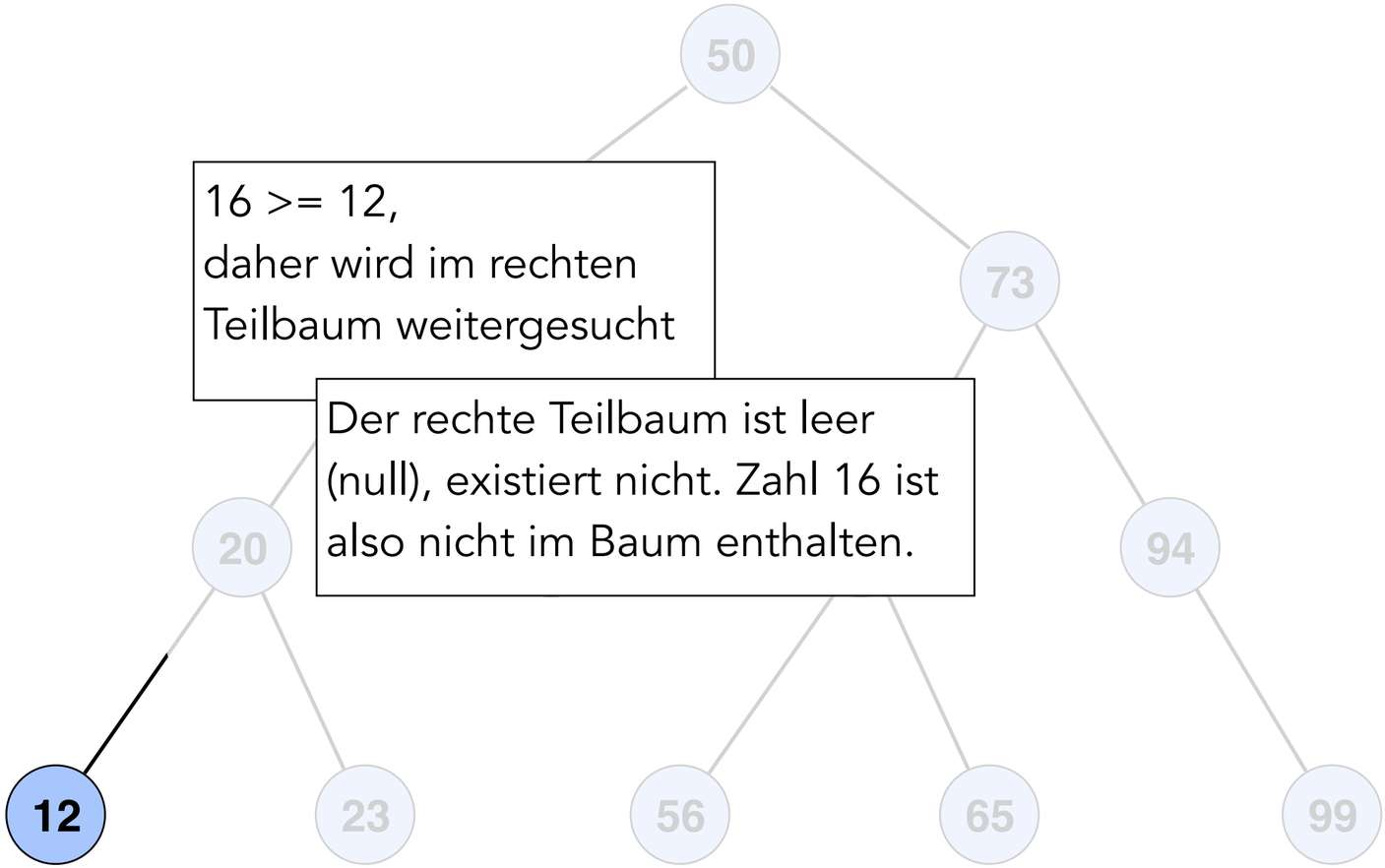
6.2 Die binäre Suche: Exkurs binäre Suchbäume

Suchen nach der 16



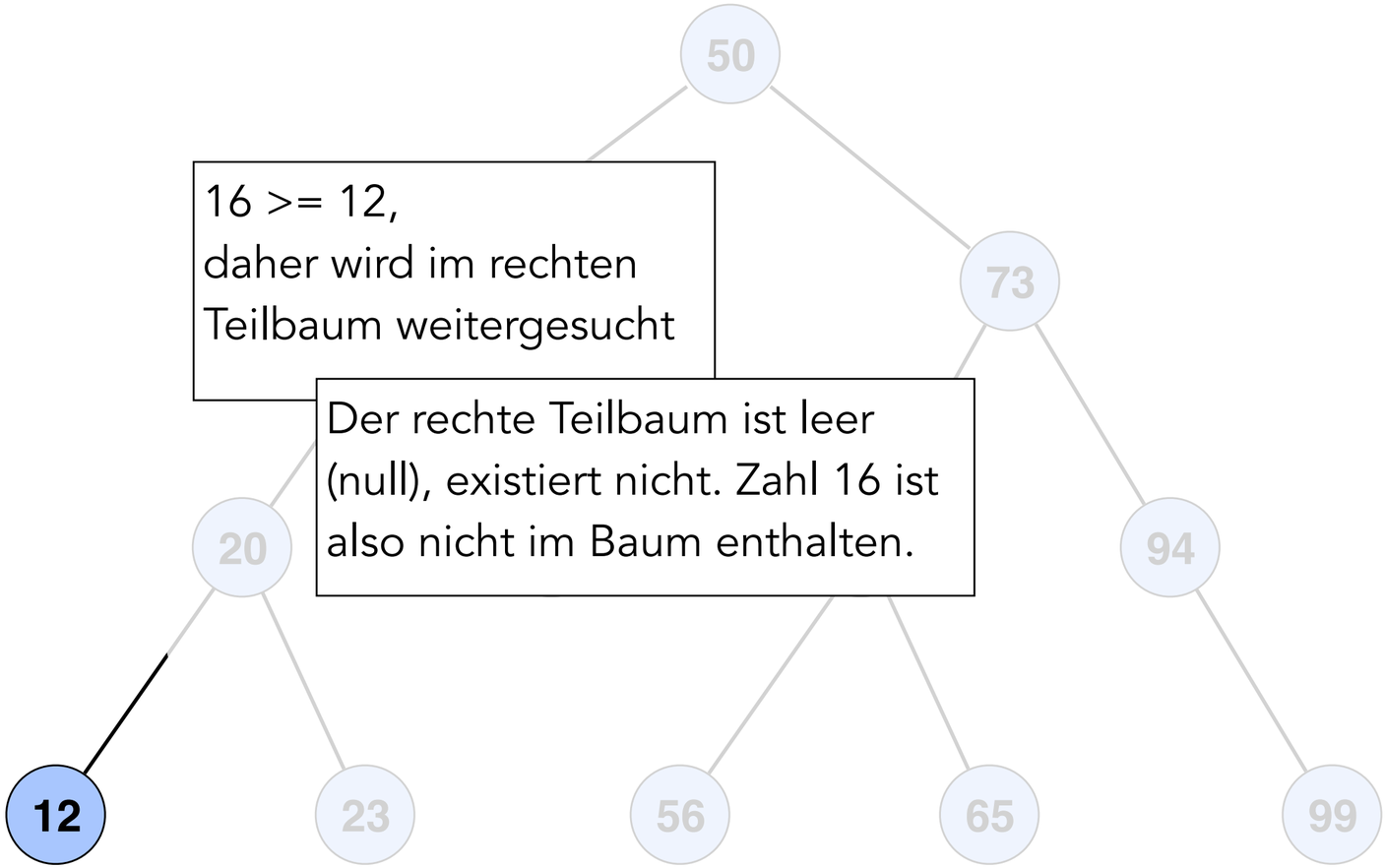
6.2 Die binäre Suche: Exkurs binäre Suchbäume

Suchen nach der 16



6.2 Die binäre Suche: Exkurs binäre Suchbäume

Suchen nach der 16

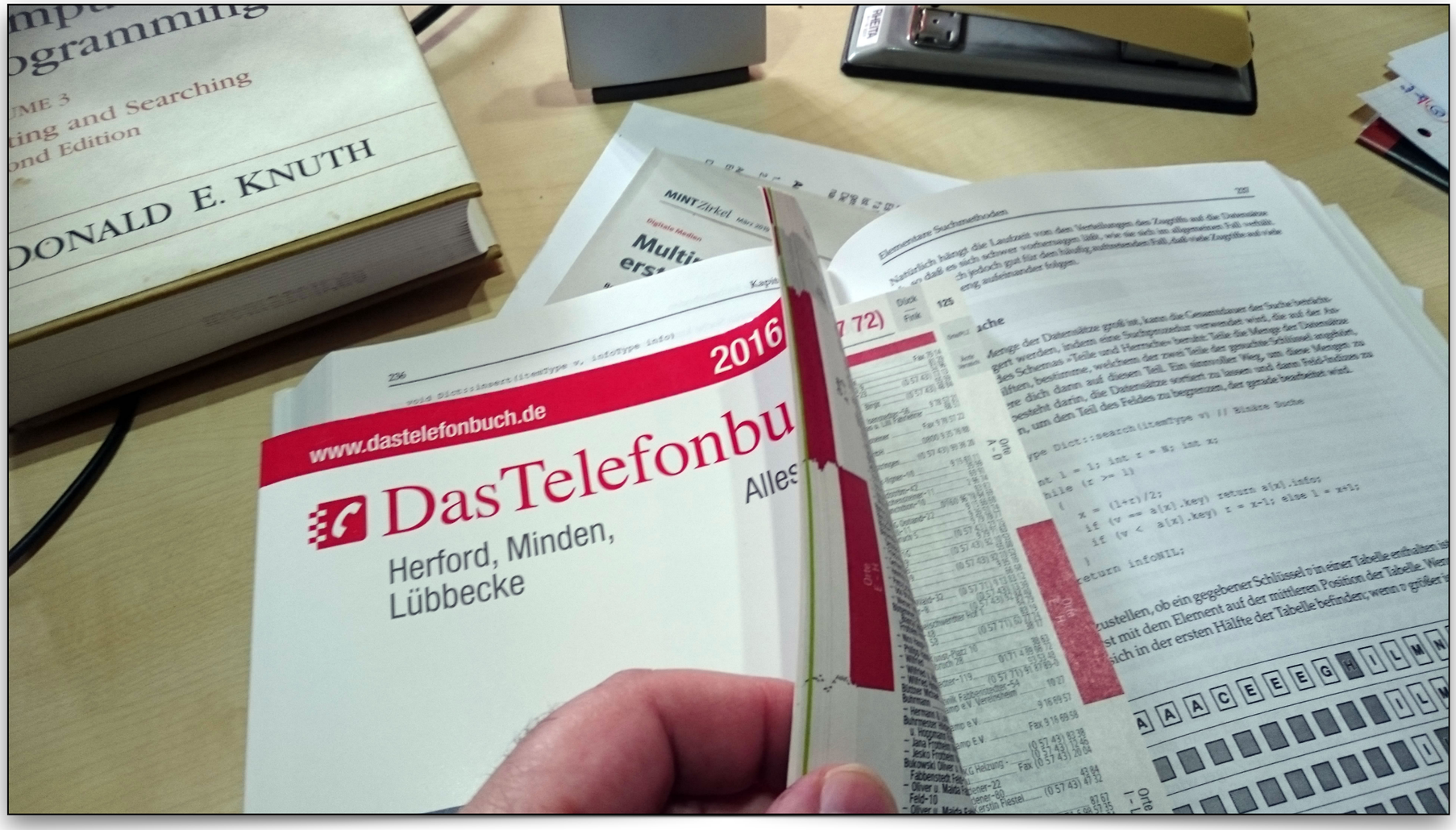


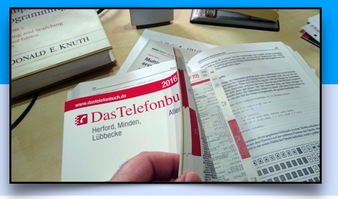
Zeitverhalten:
O(log N)*

*falls der Baum ausgeglichen ist.

6.3.1 Die Sprungsuche / jump search

Suchen nach einer Telefonnummer





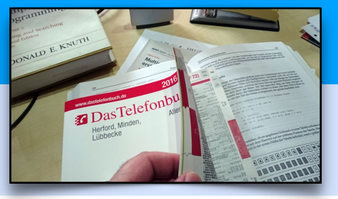
6.3.1 Die Sprungsuche / jump search

Suchen nach einer Telefonnummer

Ein sortierter Array aus 30 int-Zahlen. Gesucht wird nach der **40**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

Start der Suche bei Index 0.
2 < 40



6.3.1 Die Sprungsuche / jump search

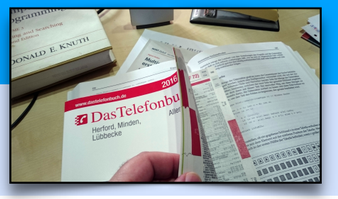
Suchen nach einer Telefonnummer

Ein sortierter Array aus 30 int-Zahlen. Gesucht wird nach der **40**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

Es wird 5 Element weiter gesprungen

12 < 40



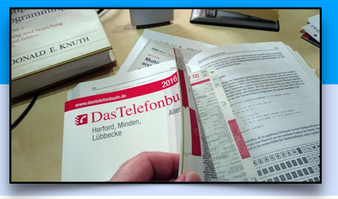
6.3.1 Die Sprungsuche / jump search

Suchen nach einer Telefonnummer

Ein sortierter Array aus 30 int-Zahlen. Gesucht wird nach der **40**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

Es wird 5 Element weiter gesprungen
22 < 40



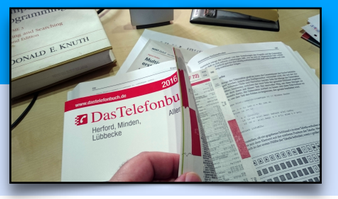
6.3.1 Die Sprungsuche / jump search

Suchen nach einer Telefonnummer

Ein sortierter Array aus 30 int-Zahlen. Gesucht wird nach der **40**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

Es wird 5 Element weiter gesprungen
32 < 40



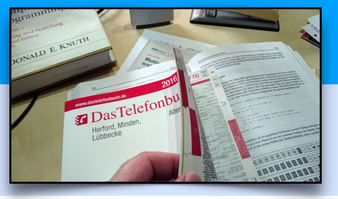
6.3.1 Die Sprungsuche / jump search

Suchen nach einer Telefonnummer

Ein sortierter Array aus 30 int-Zahlen. Gesucht wird nach der **40**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

Es wird 5 Element weiter gesprungen
42 > 40



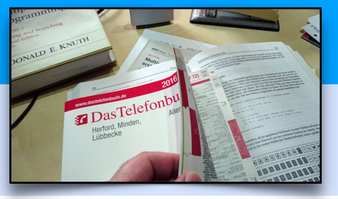
6.3.1 Die Sprungsuche / jump search

Suchen nach einer Telefonnummer

Ein sortierter Array aus 30 int-Zahlen. Gesucht wird nach der **40**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

42 > 40
Zurückspringen bis 40 gefunden wurde oder
bis eine Zahl < 40 gefunden wurde.



6.3.1 Die Sprungsuche / jump search

Suchen nach einer Telefonnummer

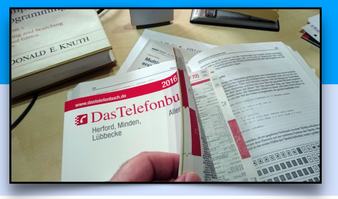
Ein sortierter Array aus 30 int-Zahlen. Gesucht wird nach der **40**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

42 > 40
Zurückspringen bis 40 gefunden wurde oder
bis eine Zahl < 40 gefunden wurde.

Zeitverhalten:
O(√N)

*Größe des Sprungintervalls = √N
Bei 30 Zahlen also 5 oder 6



6.3.1 Die Sprungsuche / jump search

Suchen nach einer Telefonnummer

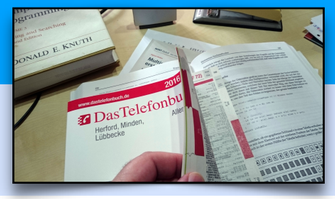
Ein sortierter Array aus 30 int-Zahlen. Gesucht wird nach der **40**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

Jump search ist also schneller als lineare Suche, aber langsamer als binäre Suche, weil in dem gefundenen Intervall linear gesucht wird.

Zeitverhalten:
O(√N)

*Größe des Sprungintervalls = √N
Bei 30 Zahlen also 5 oder 6



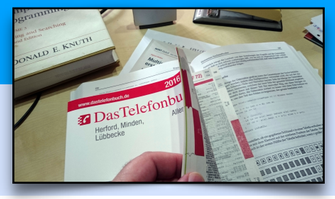
6.3.1 Die Sprungsuche / jump search

Zwei-Ebenen-Sprungsuche

Bei einem Array der Größe 1.000.000 wäre die optimale Sprungweite m = 1000.
Statt jetzt den gefundenen Zielabschnitt linear oder binär zu durchsuchen, bietet sich hierfür eine weitere Sprungsuche an. Man spricht dann von einer ***Zwei-Ebenen-Sprungsuche***.

Average case für **außen jump / innen jump** bei 1.000.000 Zahlen: ca. 32,5 + 32,5 = 64 - 65 Vergleiche

Average case für **außen jump / innen binär** bei 1.000.000 Zahlen: ca. 32,5 + 8,9 = 41 - 42 Vergleiche



6.3.1 Die Sprungsuche / jump search

Zwei-Ebenen-Sprungsuche

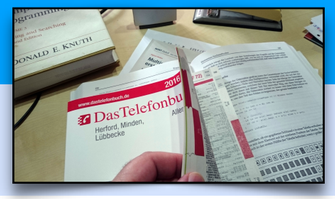
Bei einem Array der Größe 1.000.000 wäre die optimale Sprungweite m = 1000.
Statt jetzt den gefundenen Zielabschnitt linear oder binär zu durchsuchen, bietet sich hierfür eine weitere Sprungsuche an. Man spricht dann von einer ***Zwei-Ebenen-Sprungsuche***.

Average case für **außen jump / innen jump** bei 1.000.000 Zahlen: ca. 32,5 + 32,5 = 64 - 65 Vergleiche

Average case für **außen jump / innen binär** bei 1.000.000 Zahlen: ca. 32,5 + 8,9 = 41 - 42 Vergleiche

Zum Vergleich: **Reine binäre Suche** bei 1.000.000 Zahlen: ca. 19 - 20 Vergleiche.

Wieso wird dann überhaupt jump search eingesetzt?

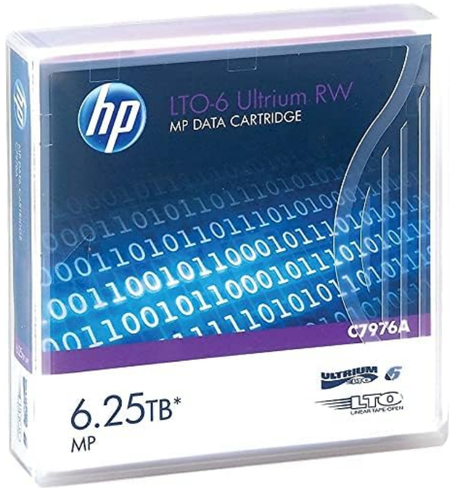


6.3.1 Die Sprungsuche / jump search

Anwendung von Jump search

Jump search ist zwar langsamer als binäre Suche, kann aber dort eingesetzt werden, wo binäre Suche nicht möglich ist:

- Dateien auf Festplatten oder SSDs
- Daten auf Bandlaufwerken
- Große externe Datenstrukturen



Eine aktuelle Datenkassette

Ein aktuelles Bandlaufwerk

6.3.2 Die Indexsuche / index search

Grundprinzip

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

6.3.2 Die Indexsuche / index search

Grundprinzip

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

1. Element >= 10

1. Element >= 20

1. Element >= 30

1. Element >= 40

1. Element >= 50

1. Element >= 60

6.3.2 Die Indexsuche / index search

Grundprinzip

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

1. Element >= 10

1. Element >= 20

1. Element >= 30

1. Element >= 40

1. Element >= 50

1. Element >= 60

0	1	2	3	4	5
4	9	14	19	24	29

6.3.2 Die Indexsuche / index search

Suche nach der 45

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

1. Element >= 10

1. Element >= 20

1. Element >= 30

1. Element >= 40

1. Element >= 50

1. Element >= 60

0	1	2	3	4	5
4	9	14	19	24	29

Phase 1: Suche nach dem richtigen Intervall

6.3.2 Die Indexsuche / index search

Suche nach der 45

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

1. Element >= 10

1. Element >= 20

1. Element >= 30

1. Element >= 40

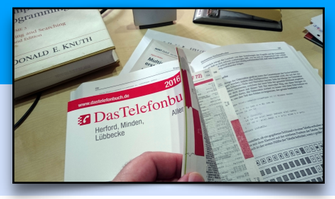
1. Element >= 50

1. Element >= 60

0	1	2	3	4	5
4	9	14	19	24	29

Phase 1: Suche nach dem richtigen Intervall

Phase 2: Suche in dem Intervall



6.3.2 Die Indexsuche / index search

Zeitverhalten

Berechnung des Zeitverhaltens bei einem Array der Größe 1.000.000

Phase 1: Binäre Suche im Indexarray (Größe 999) → $\log_2(999) = 9{,}97$, also ca. 10 Vergleiche

Phase 2: Binäre Suche im Zielintervall (durchschnittliche Größe ca. 1000) → $\log_2(1000) = 9{,}97$, 10 Vergleiche

Average case für Indexsuche **binär/binär**: 20 Vergleiche bei 1.000.000 Zahlen

Reine binäre Suche: log2(1000000) =19,932 = 20.

6.3.2 Die Indexsuche / index search

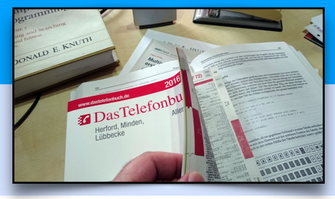
Hauptanwendungen der Indexsuche: Externe Datenspeicher.

In der ersten Phase wird mithilfe eines Index bestimmt, **welcher Datenblock** das gesuchte Element enthalten kann und **wo sich dieser Block auf dem Datenträger befindet**.

In der zweiten Phase wird der entsprechende Datenblock in den Arbeitsspeicher geladen und **dort weiter durchsucht** (z. B. linear oder binär).

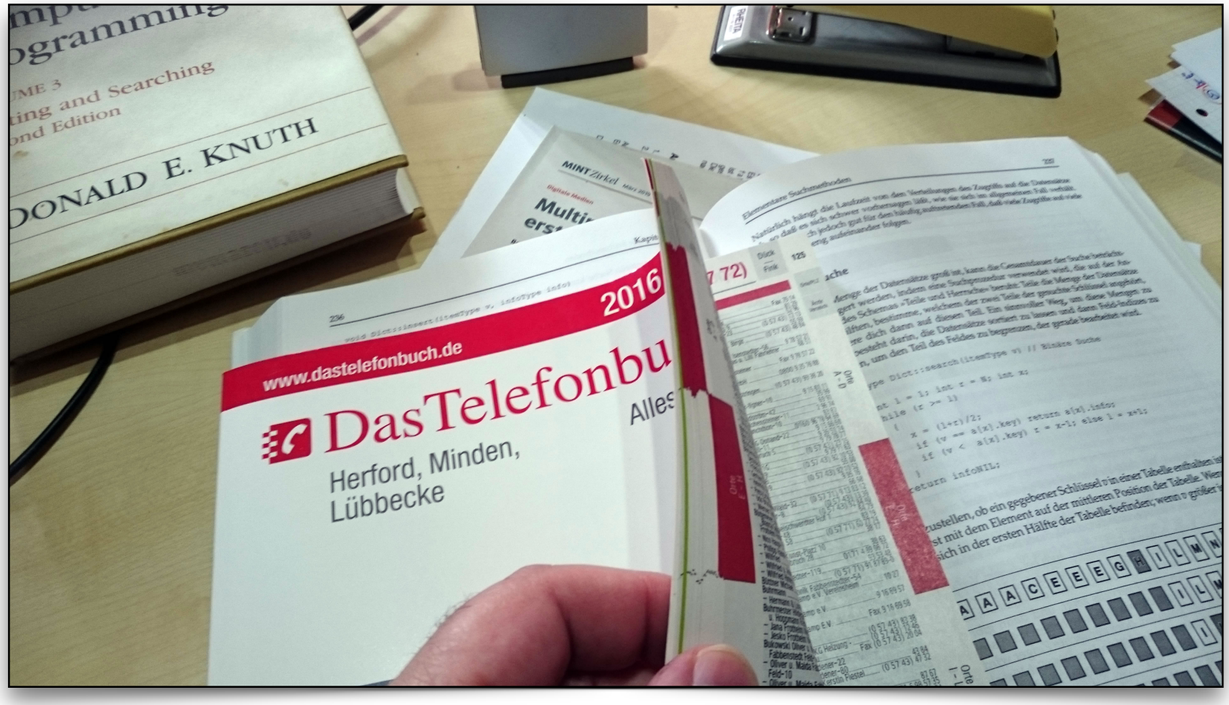
Ein einziger Festplattenzugriff oder Bandzugriff ist um Größenordnungen aufwendiger als 1000 Vergleiche im Arbeitsspeicher.

Datenbanken arbeiten ähnlich, die Indices der eigentlichen Daten werden in einer Indexdatei gespeichert.

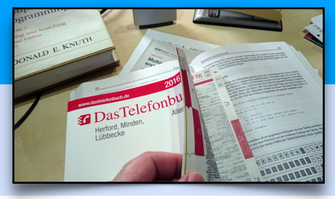


6.3.3 Die Interpolationssuche / interpolation search

Grundprinzip

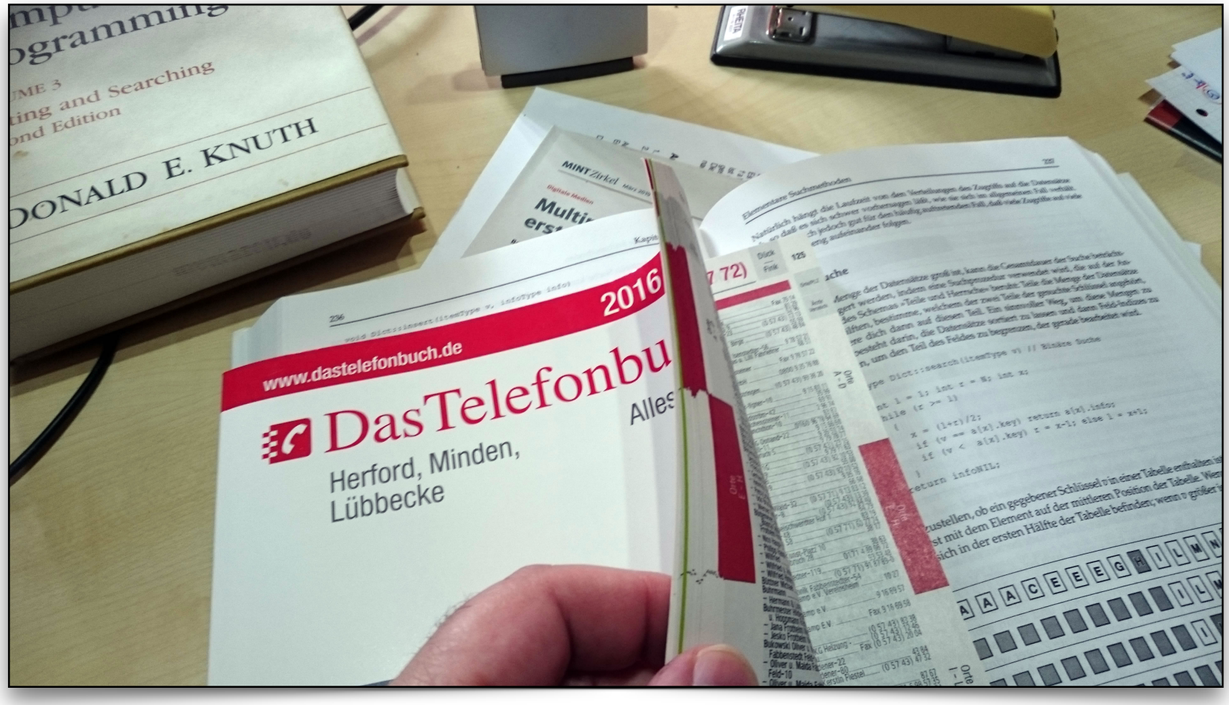


Die Interpolationssuche ist vergleichbar mit der Suche nach einer Telefonnummer in einem Telefonbuch oder nach einem Buch in einem alphabetisch nach Autoren geordnetem Bücherregal.

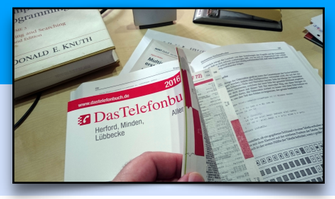


6.3.3 Die Interpolationssuche / interpolation search

Grundprinzip



Man schätzt ab, wo ungefähr sich der Eintrag befinden könnte, und fängt dann dort mit der Suche an.



6.3.3 Die Interpolationssuche / interpolation search

Algorithmus

```
public int getIndex(int[] array, int suchzahl)
{
   int links = 0; int rechts = array.length - 1;

   while (links <= rechts)
   {
      int pos = links
              + (suchzahl - array[links])
              * (rechts - links)
              / (array[rechts] - array[links]);

      if (array[pos] == suchzahl)
        return pos;
      else if (array[pos] < suchzahl)
        links = pos + 1;
      else
        rechts = pos - 1;
  }

  return -1;
}
```

6.3.3 Die Interpolationssuche / interpolation search

Algorithmus

```
public int getIndex(int[] array, int suchzahl)
{
   int links = 0; int rechts = array.length - 1;

   while (links <= rechts)
   {
      int pos = links
              + (suchzahl - array[links])
              * (rechts - links)
              / (array[rechts] - array[links]);

      if (array[pos] == suchzahl)
        return pos;
      else if (array[pos] < suchzahl)
        links = pos + 1;
      else
        rechts = pos - 1;
  }

  return -1;
}
```

Abschätzung der Such-Position

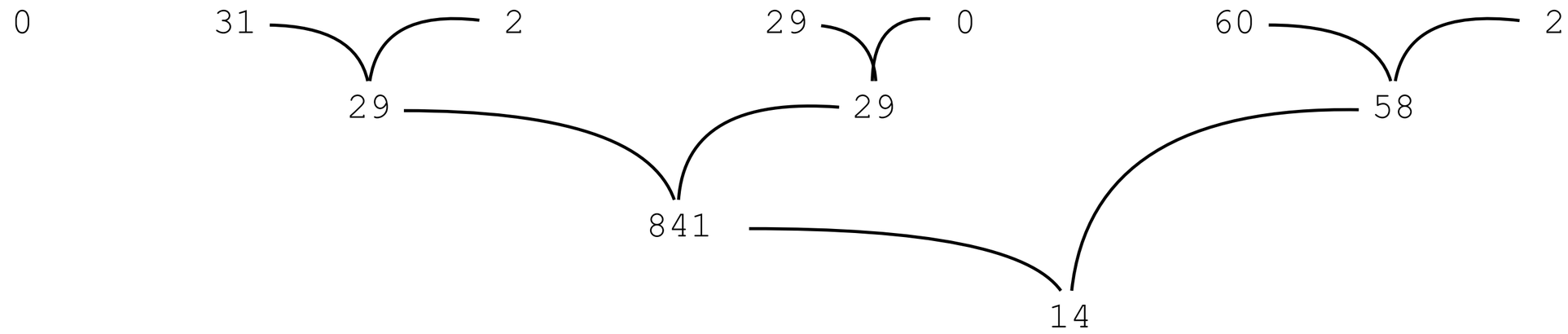
Verkleinerung des Such-Intervalls

(ähnlich wie bei der binären Suche)

6.3.3 Die Interpolationssuche / interpolation search

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

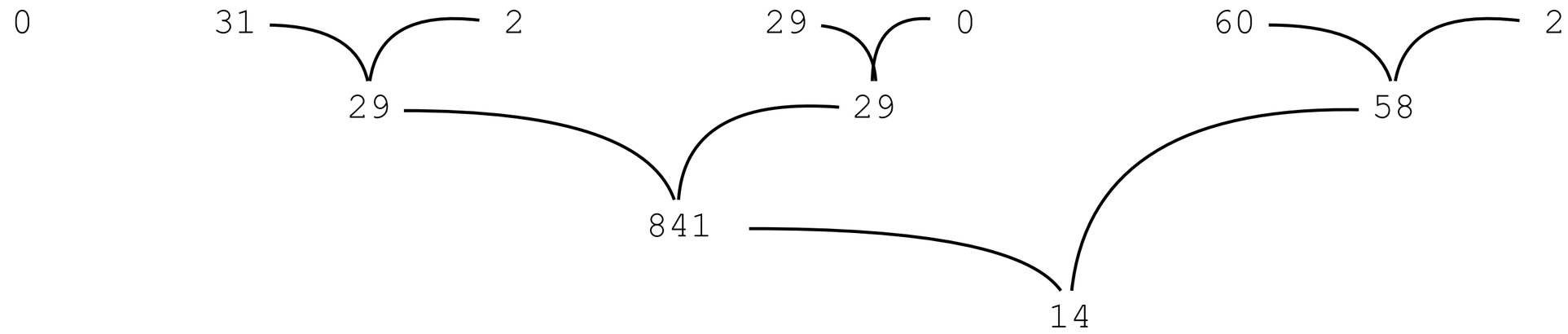
```
int pos = links + (suchzahl - array[links])* (rechts - links) / (array[rechts] - array[links]);
```

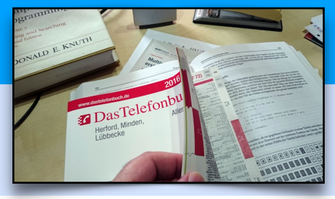


6.3.3 Die Interpolationssuche / interpolation search

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2	4	6	8	11	12	14	15	18	21	22	25	26	27	31	32	34	35	39	40	41	45	46	48	50	52	54	55	58	60

```
int pos = links + (suchzahl - array[links])* (rechts - links) / (array[rechts] - array[links]);
```





6.3.4 Zusammenfassung der drei Verfahren

Sprungsuche:
Das Array wird blockweise durchsucht: Man springt in festen Schritten vorwärts und sucht innerhalb des passenden Blocks anschließend linear.

Indexsuche:
Ein separater Index zeigt auf grobe Positionen im Datenbestand; gesucht wird zuerst im Index und danach gezielt im zugehörigen Datenblock.

Interpolationssuche:
Die Suchposition wird aus dem Wert selbst geschätzt: Je nach Zahlenwert springt man direkt dorthin, wo der Eintrag wahrscheinlich liegt.