6. Suchalgorithmen

6.1 Allgemeines zum Suchen

Typische Situationen:

1. Die Daten sind unsortiert.
2. Die Daten sind sortiert, das Suchkriterium entspricht aber nicht dem Sortierkriterium.
3. Die Daten sind sortiert und das Suchkriterium entspricht dem Sortierkriterium.
4. Die Daten sind suchfreundlich sortiert, das Suchkriterium entspricht dem Sortierkriterium.

6. Suchalgorithmen

6.1 Allgemeines zum Suchen

6.1.1 Vier Beispiele aus dem Alltag

Interpolationssuche

Suche im Bücherregal nach "Karl Melcher".

Man schätzt ab, ob "M" ungefähr steht und beginnt dort mit der Suche.

Suche im Telefonbuch nach "Hans Xaver"

Man fängt hinten an zu suchen.

6. Suchalgorithmen

6.1 Allgemeines zum Suchen

6.1.1 Vier Beispiele aus dem Alltag

Indexsuche

Suche im Register eines Informatikbuchs nach "Suchverfahren"

Suche in der digitalen Kartei einer Bibliothek nach einem Autoren.

6. Suchalgorithmen

6.1 Allgemeines zum Suchen

6.1.1 Vier Beispiele aus dem Alltag

Binäre Suche

Suche nach einer nicht-löschbaren Datei im Papierkorb des Rechners. Welche der 1.300 Dateien ist defekt?

6. Suchalgorithmen

6.1 Allgemeines zum Suchen

6.1.1 Vier Beispiele aus dem Alltag

Sequentielle Suche

Suche in der Schallplattensammlung des Vaters nach "The Luftschiff". Die Platten sind aber nach Kaufdatum sortiert.

6. Suchalgorithmen

6.2 Sequentielle Suche

6.2.1 Zeitverhalten

Günstigster Fall (Best Case)	Ungünstigster Fall (Worst Case)	Durchschnittlicher Fall (Average Case)

6. Suchalgorithmen

6.2 Sequentielle Suche

6.2.1 Zeitverhalten

Günstigster Fall
(Best Case)

Das gesuchte Element befindet sich an erster Stelle, daher ist nur ein Vergleich notwendig: **S = 1**.

Ungünstigster Fall
(Worst Case)

Durchschnittlicher Fall
(Average Case)

6. Suchalgorithmen

6.2 Sequentielle Suche

6.2.1 Zeitverhalten

Günstigster Fall
(Best Case)

Das gesuchte Element befindet sich an erster Stelle, daher ist nur ein Vergleich notwendig: **S = 1**.

Ungünstigster Fall
(Worst Case)

Das gesuchte Element befindet sich an letzter Stelle, daher sind n Vergleiche notwendig: **S = n**.

Durchschnittlicher Fall
(Average Case)

6. Suchalgorithmen

6.2 Sequentielle Suche

6.2.1 Zeitverhalten

Günstigster Fall
(Best Case)

Das gesuchte Element befindet sich an erster Stelle, daher ist nur ein Vergleich notwendig: **S = 1**.

Ungünstigster Fall
(Worst Case)

Das gesuchte Element befindet sich an letzter Stelle, daher sind n Vergleiche notwendig: **S = n**.

Durchschnittlicher Fall
(Average Case)

Das gesuchte Element befindet mit gleicher Wahrscheinlichkeit an jeder Position: **S = (n+1)/2**

6. Suchalgorithmen

6.2 Sequentielle Suche

6.2.1 Zeitverhalten

Günstigster Fall
(Best Case)

Das gesuchte Element befindet sich an erster Stelle, daher ist nur ein Vergleich notwendig: **S = 1**.

Ungünstigster Fall
(Worst Case)

Das gesuchte Element befindet sich an letzter Stelle, daher sind n Vergleiche notwendig: **S = n**.

Durchschnittlicher Fall
(Average Case)

Das gesuchte Element befindet mit gleicher Wahrscheinlichkeit an jeder Position: **S = (n+1)/2**

Zeitkomplexität ist O(n).

Durchschnittlicher Fall (Average case): S = (n+1)/2

In einer Liste mit sechs Elementen wird ein Element nach einem Vergleich gefunden, ein Element nach zwei Vergleichen und so weiter: S = (1+2+3+4+5+6) / 6 = 21/6 = 3,5 = **(6+1)/2 = (n+1)/2.**

6. Suchalgorithmen

6.2 Sequentielle Suche

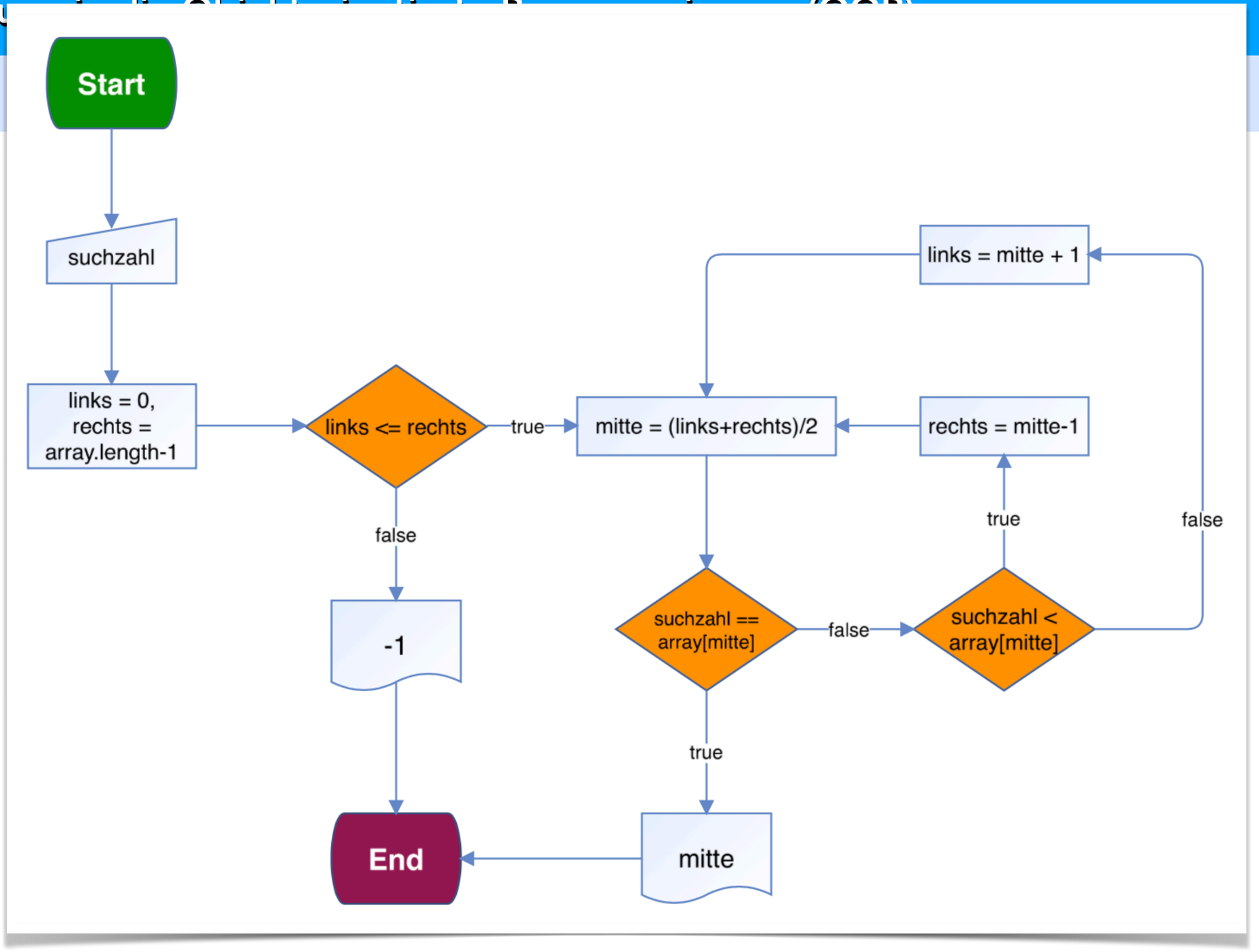
6.2.2 Implementation

```
public int getIndex(int[] zahlen, int suchzahl)
{
   for (int i = 0; i < zahlen.length; i++)
      if (zahlen[i] == suchzahl)
         return i;
   return -1;
}
```

6. Suchalgorithmen

6.3 Binäre Suche

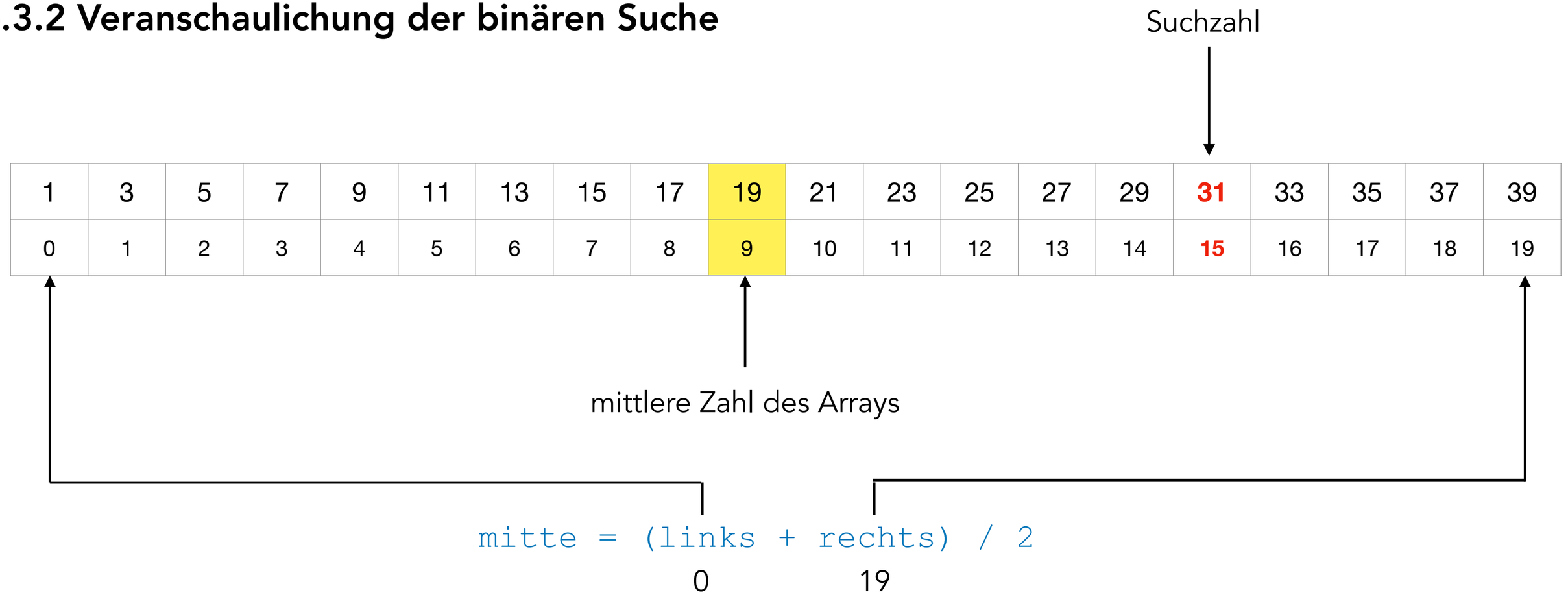
6.3.1 Der Algorithmus



6. Suchalgorithmen

6.3 Binäre Suche

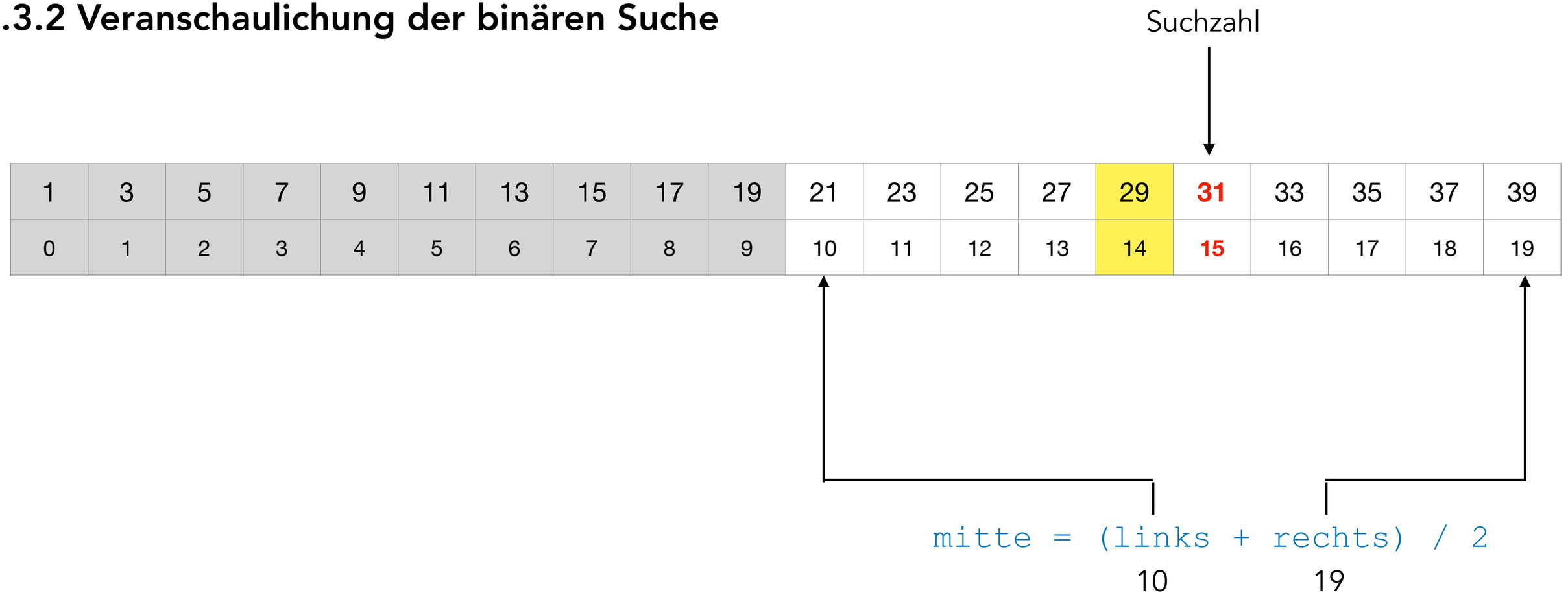
6.3.2 Veranschaulichung der binären Suche



6. Suchalgorithmen

6.3 Binäre Suche

6.3.2 Veranschaulichung der binären Suche



6. Suchalgorithmen

6.3 Binäre Suche

6.3.2 Veranschaulichung der binären Suche

6. Suchalgorithmen

6.3 Binäre Suche

6.3.2 Veranschaulichung der binären Suche

6. Suchalgorithmen

6.3 Binäre Suche

6.3.3 Quantitative Analyse der binären Suche

Quelltext einer Methode zur binären Suche nach der Position der Suchzahl in dem übergebenen Array `a`.

```
public int sucheIndexBinaer(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 mitte;

       if (suchzahl < a[mitte])
          rechts = mitte - 1;
       else
          links = mitte + 1;
   }

   return -1;
}
```

6. Suchalgorithmen

6.3 Binäre Suche

6.3.3 Quantitative Analyse der binären Suche

Quelltext einer ***rekursiven*** Methode zur binären Suche nach der Position der Suchzahl in dem übergebenen Array `a`.

```
public int vergleicheBinaer(int[] a, int suchzahl)
{
   return rekursiv(a, suchzahl, 0, a.length - 1);
}

private int rekursiv(int[] a, int suchzahl, int links, int rechts)
{
   if (links <= rechts)
   {
      int mitte = (links + rechts) / 2;

      if (a[mitte] == suchzahl)
         return mitte;

      if (suchzahl < a[mitte])
         return rekursiv(a, suchzahl, links, mitte - 1);
      else
         return rekursiv(a, suchzahl, mitte + 1, rechts);
   }

   return -1;
}
```

6. Suchalgorithmen

6.3 Binäre Suche

6.3.3 Quantitative Analyse der binären Suche

Modifizierter Quelltext einer Methode zur binären Suche.

Die Zahl der Vergleiche wird ermittelt.

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

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

        v++;
        if (a[mitte] == suchzahl) return v;

        v++;
        if (suchzahl < a[mitte])
           rechts = mitte - 1;
        else
           links = mitte + 1;
    }

    return v;
}
```

6. Suchalgorithmen

6.3 Binäre Suche

6.3.3 Quantitative Analyse der binären Suche

Modifizierter Quelltext einer Methode zur linearen Suche.

Die Zahl der Vergleiche wird ermittelt.

```
public int vergleicheLinear(int[] a, int suchzahl)
 {
     int v = 0;

     for (int i = 0; i < a.length; i++)
     {
         v++;
         if (a[i] == suchzahl)
         {
             return v;
         }
     }

     return v;
 }
```

6. Suchalgorithmen

6.3 Binäre Suche

6.3.3 Quantitative Analyse der binären Suche

Ausgabe eines Testprogramms

```
Lineare Suche für 64 Zahlen = 2608 Vergleiche
Binaere Suche für 64 Zahlen = 353 Vergleiche
Verhältnis Linear/Binär =         7,39

Lineare Suche für 128 Zahlen = 10336 Vergleiche
Binaere Suche für 128 Zahlen = 833 Vergleiche
Verhältnis Linear/Binär =        12,41

Lineare Suche für 256 Zahlen = 41152 Vergleiche
Binaere Suche für 256 Zahlen = 1921 Vergleiche
Verhältnis Linear/Binär =        21,42

Lineare Suche für 512 Zahlen = 164224 Vergleiche
Binaere Suche für 512 Zahlen = 4353 Vergleiche
Verhältnis Linear/Binär =        37,73

Lineare Suche für 1024 Zahlen = 656128 Vergleiche
Binaere Suche für 1024 Zahlen = 9729 Vergleiche
Verhältnis Linear/Binär =        67,44

Lineare Suche für 2048 Zahlen = 2622976 Vergleiche
Binaere Suche für 2048 Zahlen = 21505 Vergleiche
Verhältnis Linear/Binär =       121,97

Lineare Suche für 4096 Zahlen = 10488832 Vergleiche
Binaere Suche für 4096 Zahlen = 47105 Vergleiche
Verhältnis Linear/Binär =       222,67
```

6. Suchalgorithmen

6.3 Binäre Suche

6.3.3 Quantitative Analyse der binären Suche

Übung 6.1

Erstellen Sie ein solches Testprogramm!

Übung 6.2

Ermitteln Sie, ob die rekursive binäre Suche weniger Vergleiche benötigt als die nicht-rekursive Variante.

6. Suchalgorithmen

6.3 Binäre Suche

6.3.3 Quantitative Analyse der binären Suche

Übung 6.3

Eine KI hat den Quelltext der nicht-rekursiven Variante analysiert und meint, dass folgender Quelltext noch besser ist:

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

    while (links <= rechts) {
        int mitte = (links + rechts) / 2;
        v++;
        if      (suchzahl < a[mitte]) rechts = mitte - 1;
        else if (suchzahl > a[mitte]) links  = mitte + 1;
        else return v;
    }

    return v;
}
```

Vergleichen Sie diesen Quelltext mit dem vorherigen nicht-optimierten.

Analysieren Sie, ob die binäre Suche dadurch noch realistischer dargestellt wird.

6. Suchalgorithmen

6.3 Binäre Suche

6.3.3 Quantitative Analyse der binären Suche

Übung 6.4

Analysieren Sie die folgende Tabelle:

N	64	128	256	512	1024	2048	4096
Vergleiche V bei binärer Suche	353	883	1921	4353	9729	21505	47105
$\log_2(N)$	6	7	8	9	10	11	12
Quotient V / N	5,52	6,90	7,50	8,50	9,50	10,50	11,50

Begründen Sie mit den Daten aus der Tabelle, wieso das Zeitverhalten der binären Suche durch **O(log(N))** charakterisiert werden kann.

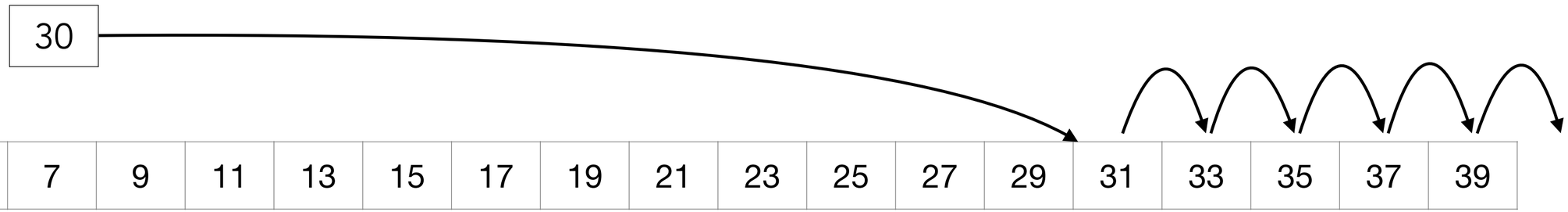
6. Suchalgorithmen

6.3 Binäre Suche

6.3.4 Binäre Suchbäume

1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	30	33	35	37	39

Daten sind sortiert → binäre Suche recht einfach.



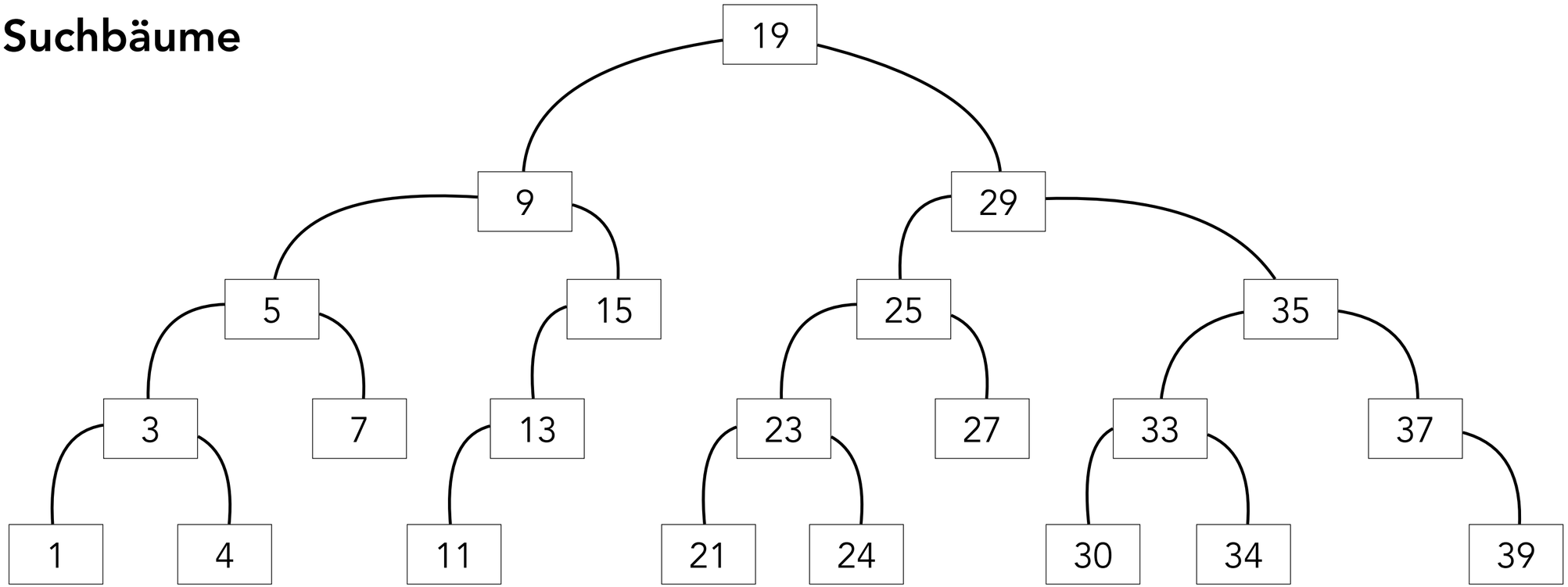
Einfügen neuer Daten→ sehr aufwendig.

Große Datenbanken können nicht nach jeder Bearbeitung neu sortiert werden.

6. Suchalgorithmen

6.3 Binäre Suche

6.3.4 Binäre Suchbäume



1	3	4	5	7	9	11	13	15	21	23	24	25	27	29	30	33	34	35	37	39

6. Suchalgorithmen

6.3 Binäre Suche

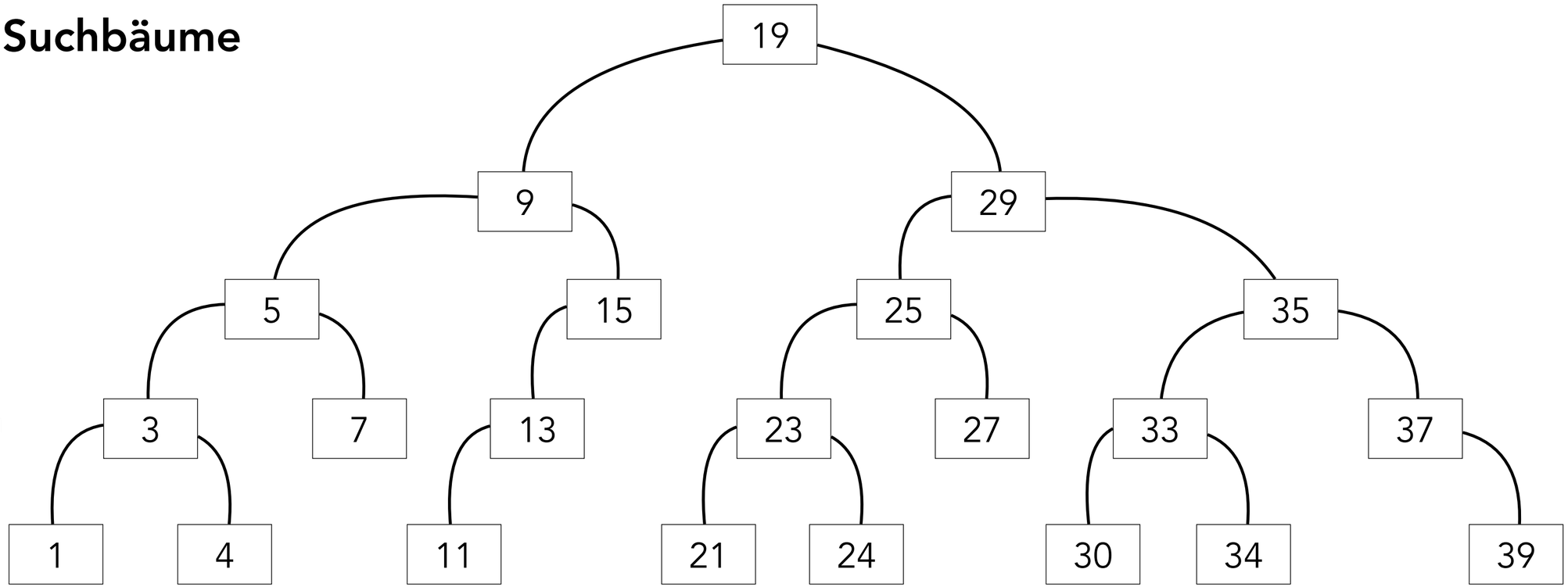
6.3.4 Binäre Suchbäume

Suchzahl: 33

Nach **vier** Vergleichen gefunden -

wie bei der binären Suche in einem Array.

O(log(N))



1	3	4	5	7	9	11	13	15	21	23	24	25	27	29	30	33	34	35	37	39

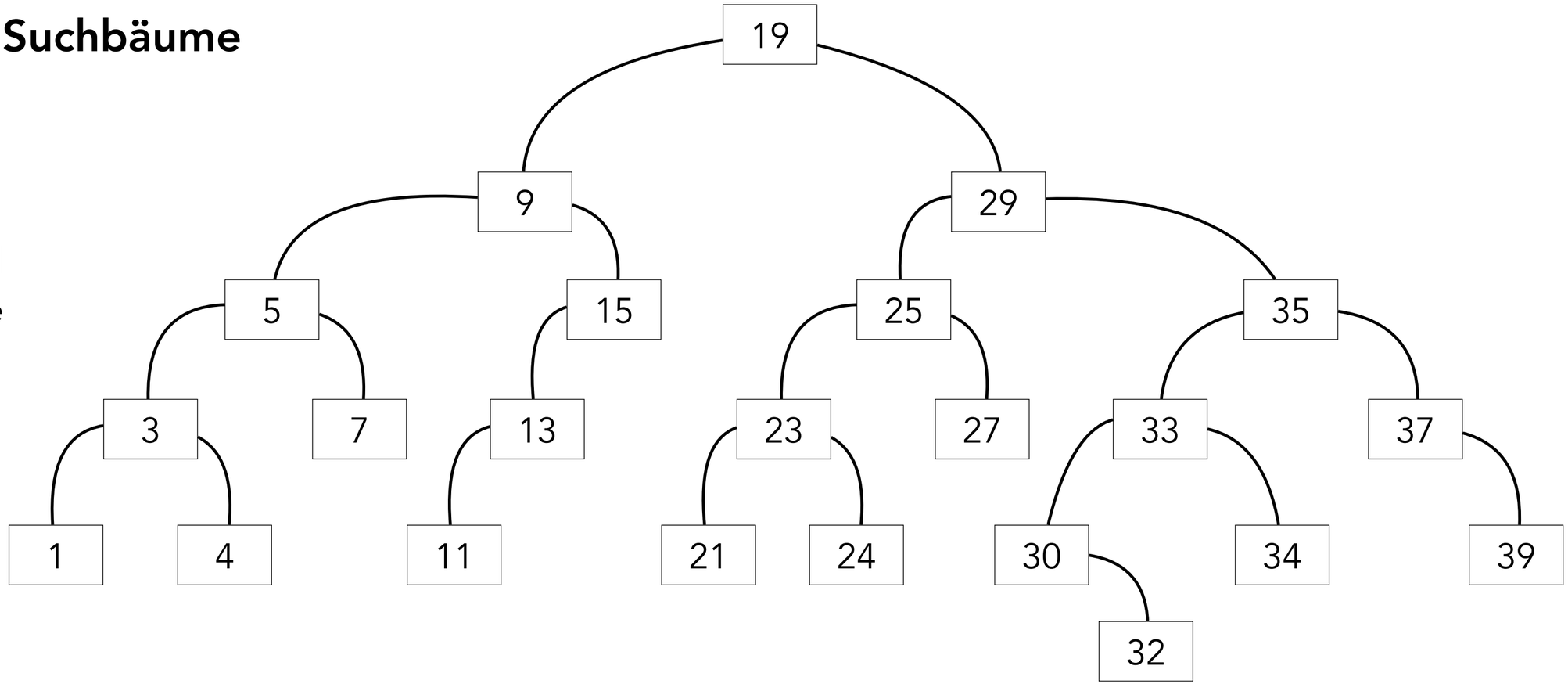
6. Suchalgorithmen

6.3 Binäre Suche

6.3.4 Binäre Suchbäume

Einfügen: 32

Nach **sechs** Vergleichen wird die Einfügestelle gefunden.



6. Suchalgorithmen

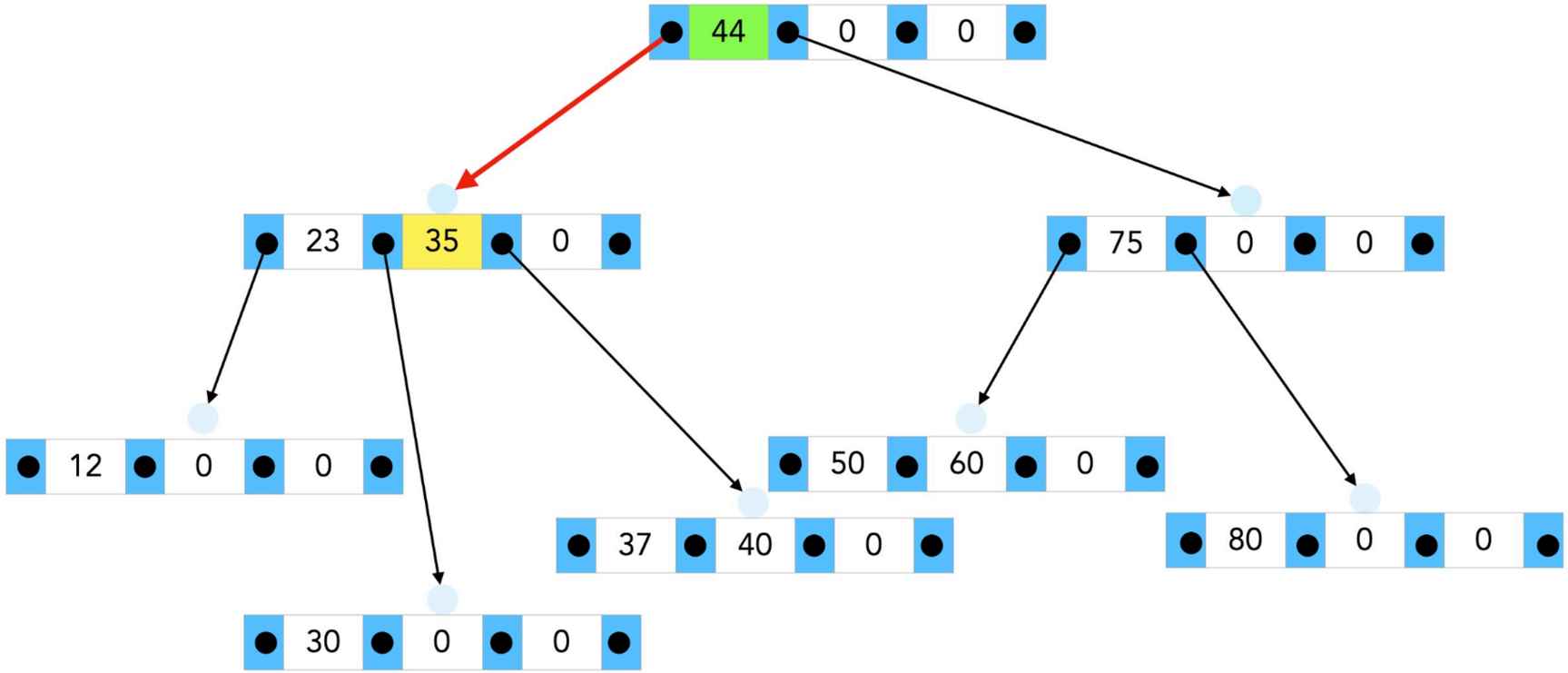
6.3 Binäre Suche

6.3.5 B-Bäume

B-Bäume werden für Indexdateien von Datenbanken eingesetzt.

Jeder Knoten kann N Elemente und N+1 Nachfolger enthalten.

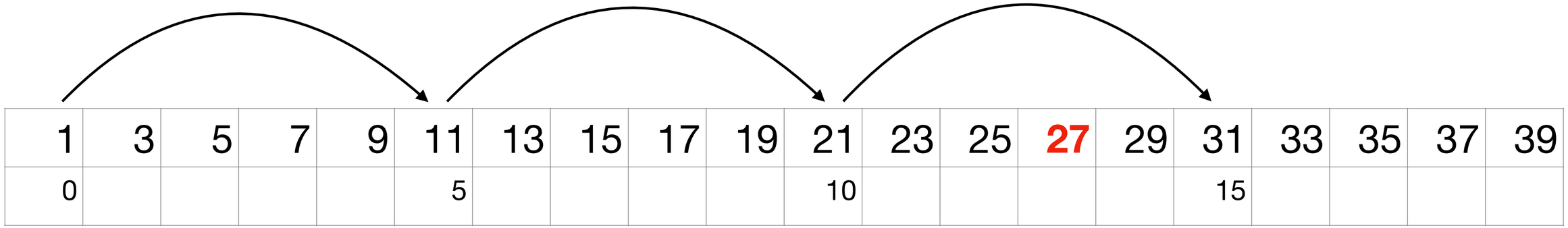
Einzelheiten dazu auf dieser Webseite und im Kurs **Datenstrukturen**.



6. Suchalgorithmen

6.4 Weitere einfache Suchverfahren

6.4.1 Die Sprungsuche



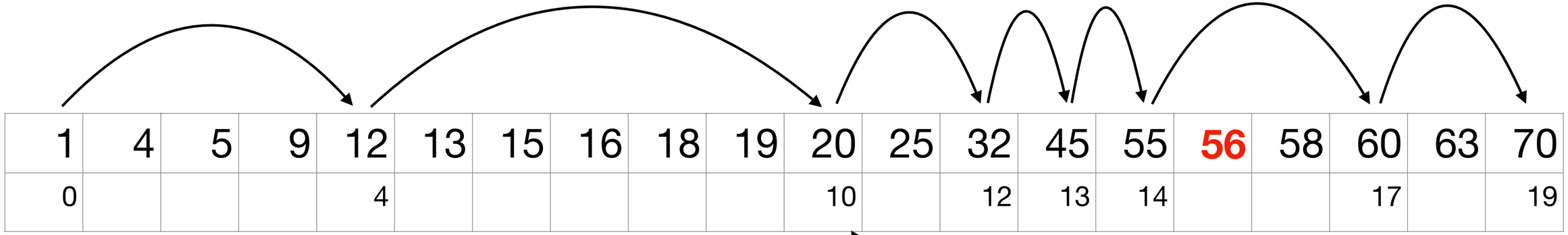
Übung 6.5

Erläutern Sie das Grundprinzip dieses Verfahrens!

6. Suchalgorithmen

6.4 Weitere einfache Suchverfahren

6.4.2 Die Indexsuche



Übung 6.6

Erläutern Sie das Grundprinzip dieses Verfahrens!

Was ist der Unterschied zur Sprungsuche?

Index-Array

6. Suchalgorithmen

6.4 Weitere einfache Suchverfahren

6.4.3 Die Interpolationssuche

1	3	5	8	11	13	15	17	18	23	25	27	28	29	30	**31**	33	38	39	41
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	**15**	16	17	18	19

Wertebereich 1 .. 41, ***Spannweite*** also 40.

Verteilt auf 20 Elemente, das sind 19 ***Intervalle***.

40 / 19 = 2,1 → **Pro Index steigt der Wert der Zahlen um 2,1.**

6. Suchalgorithmen

6.4 Weitere einfache Suchverfahren

6.4.3 Die Interpolationssuche

1	3	5	8	11	13	15	17	18	23	25	27	28	29	30	**31**	33	38	39	41
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	**15**	16	17	18	19

Wertebereich 1 .. 41, ***Spannweite*** also 40.

Verteilt auf 20 Elemente, das sind 19 ***Intervalle***.

40 / 19 = 2,1 → **Pro Index steigt der Wert der Zahlen um 2,1.**

Suche nach der 31

Berechnung der möglichen Position der Suchzahl: (31 - 1) / 2,1 = 14

Ab da lineare Suche, nach 1 Vergleich wird die 31 gefunden

6. Suchalgorithmen

6.4 Weitere einfache Suchverfahren

6.4.3 Die Interpolationssuche

1	3	5	8	11	13	15	17	18	23	25	27	28	29	30	31	33	38	39	41
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Wertebereich 1 .. 41, ***Spannweite*** also 40.

Verteilt auf 20 Elemente, das sind 19 ***Intervalle***.

40 / 19 = 2,1 → **Pro Index steigt der Wert der Zahlen um 2,1.**

Suche nach der 17

Berechnung der möglichen Position der Suchzahl: (17 - 1) / 2,1 = 7

Suchzahl wird sofort gefunden!

6. Suchalgorithmen

6.4 Weitere einfache Suchverfahren

6.4.3 Die Interpolationssuche

1	3	5	8	11	13	15	17	18	23	25	27	28	29	30	**31**	33	38	39	41
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	**15**	16	17	18	19

Wertebereich 1 .. 41, ***Spannweite*** also 40.

Verteilt auf 20 Elemente, das sind 19 ***Intervalle***.

40 / 19 = 2,1 → **Pro Index steigt der Wert der Zahlen um 2,1.**

Suche nach der 35

Berechnung der möglichen Position der Suchzahl: (35 - 1) / 2,1 = 16

Ab da lineare Suche, nach 1 Vergleich ist klar, dass 35 nicht enthalten ist.

6. Suchalgorithmen

6.4 Weitere einfache Suchverf

6.4.3 Die Interpolationssuche

Übung 6.7

Erläutern Sie die Arbeitsweise dieser Methode.

Die **Experten** unter Ihnen versuchen eine solche Interpolationssuche zu implementieren und systematisch zu testen.

```
1  public int interpolationssuche(int[] a, int suchzahl)
2  {
3      if (a == null || a.length == 0) return -1;
4      if (a.length == 1) {
5          if (a[0] == suchzahl) return 0;
6          return -1;
7      }
8      int minimum = getMinimum(a);
9      int maximum = getMaximum(a);
10     int spannweite = maximum - minimum;
11     int intervalle = a.length-1;
12     double inkrement = (double) spannweite / intervalle;
13     int wIndex = (int) ((suchzahl - minimum) / inkrement);
14
15     if (wIndex < 0) wIndex = 0;
16     if (wIndex > a.length-1) wIndex = a.length-1;
17     if (suchzahl == a[wIndex]) return wIndex;
18     if (suchzahl < a[wIndex])
19         return sucheLinearRueckwaerts(a, wIndex, suchzahl);
20     else
21         return sucheLinearVorwaerts(a, wIndex, suchzahl);
22 }
```

6. Suchalgorithmen

6.4 Weitere einfache Suchverfahren

6.4.4 Zusammenfassung

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.