8.

Binärsuche auf Arrays

Buch Mark Weiss „Data Structures & Problem Solving Using Java“ siehe:
Section 5.6.2; Seiten 208-212; 244-248; 341-342

Lernziele Kapitel 8 Binärsuche auf Arrays

- Binärsuche auf sortierten Arrays verstehen, insbesondere die Effizienz und die Anwendungsvoraussetzungen sowie iterative versus rekursive Implementierung
- Prinzip der linearen Interpolationssuche verstehen

Thema / Inhalt

Eines der kürzeren Kapitel der Vorlesung, aber ein relevantes Thema: Hat man eine (grössere) Menge von Elementen entsprechend einem Schlüsselattribut sortiert vorliegen und kann man effizient an jede Stelle gelangen (was bei einem Array der Fall ist), dann kann man in wenigen Schritten herausfinden, ob ein gesuchtes Element (das durch den Wert des Schlüsselattributs eindeutig identifiziert ist) dort abgespeichert ist oder nicht, und im positiven Fall auch gleich den Zugriff darauf liefern (z.B. in Form der Indexnummer des Arrays).

Die Idee ist einfach: Man prüft fortwährend (iterativ oder rekursiv), in welcher Hälfte des Restbereichs das Element liegen müsste – dazu sticht man in die Mitte, ermittelt den Schlüsselwert des dort befindlichen Elements und setzt dann entweder die rechte oder linke Grenze neu auf diese Mitte, je nachdem, ob man links oder rechts weitersuchen möchte – wenn man nicht zufällig das gesuchte Element gerade in der Mitte angetroffen hat! „Kollabiert" der Restbereich, dann ist das Gesuchte nicht gespeichert. Ansonsten trifft man beim mittigen Hineinstechen früher oder später (meist allerdings erst ziemlich spät...) auf das gesuchte Element.

Thema / Inhalt (2)

Da der Restbereich mit jeder Iteration halbiert wird (genau genommen oft sogar noch etwas kleiner wird, da man die zuletzt bestimmte Mitte ausschliessen kann), benötigt man nur logarithmisch viele Iterationen bzw. Intervallhalbierungen – logarithmisch bezüglich der Gesamtzahl der Elemente bzw. der Länge des Arrays. (Die Basis der Logarithmus ist natürlich 2; aber wenn man dies mit dem natürlichen Logarithmus oder dem Logarithmus zur Basis 10 abschätzen würde, dann wäre der Fehler überschaubar, denn die Logarithmen zu unterschiedlichen Basen sind ja proportional zueinander.)

Wenn der Datenbestand, in dem man sucht, statisch ist, klappt alles gut. Wenn er aber dynamisch ist, wenn also Elemente eingefügt werden oder gelöscht werden, ist die Sache a priori nicht mehr so effizient: Wenn man ein neues Element an die richtige Stelle im sortierten Array einfügt, dann muss man Platz durch Verschieben schaffen – dazu müssen viele, typischerweise ca. die Hälfte, der Elemente verschoben werden. Ähnlich ist es beim Löschen, wenn man das Array kompaktifiziert. Mit einem kleinen Trick kann man sich hier helfen, wenn Updates, also Einfügungen oder Löschungen, eher selten relativ zum Suchen vorkommen: Beim Löschen würde man nicht kompaktifizieren, sondern das gelöschte Element lediglich als ungültig markieren. Entsprechend könnte man von vornherein gleich eine gewisse Zahl von „Lücken" lassen, so dass das Verschieben beim Einfügen nur wenige nebeneinanderstehende Elemente bis zur nächsten Lücke betrifft. (Natürlich müssen dann diese Lücken bei der Intervallhalbierung bzgl. des Schlüsselattributs geeignet behandelt werden.) Von Zeit zu Zeit würde man dann das Array reorganisieren, muss diesen grösseren (und amortisierten) Aufwand aber nur selten spendieren (z.B. dann, wenn bei Beendigung einer interaktiven Anwendung die Daten aus dem Hauptspeicher sowieso auf ein permanentes Speichermedium geschrieben werden müssen).

Thema / Inhalt (3)

Treibt man die Idee einer Datenstruktur weiter, bei der nicht nur das Suchen effizient (d.h. mit logarithmischem Aufwand) möglich ist, sondern auch das Einfügen und Löschen, dann kommt man zu Bäumen – primär den sogenannten binären Suchbäumen, die es in diversen balancierten Varianten gibt, um einer Entartung durch maliziöse Update-Folgen vorzubeugen. Wenn man besonders genial ist, wird man vielleicht auch die **Heap**-Datenstruktur neu entdecken. Das ist im Rahmen dieser Vorlesung aber nicht nötig, denn wir werden in späteren Kapiteln sowohl binäre Suchbäume (Kapitel 9) als auch Heaps (Kapitel 15) behandeln.

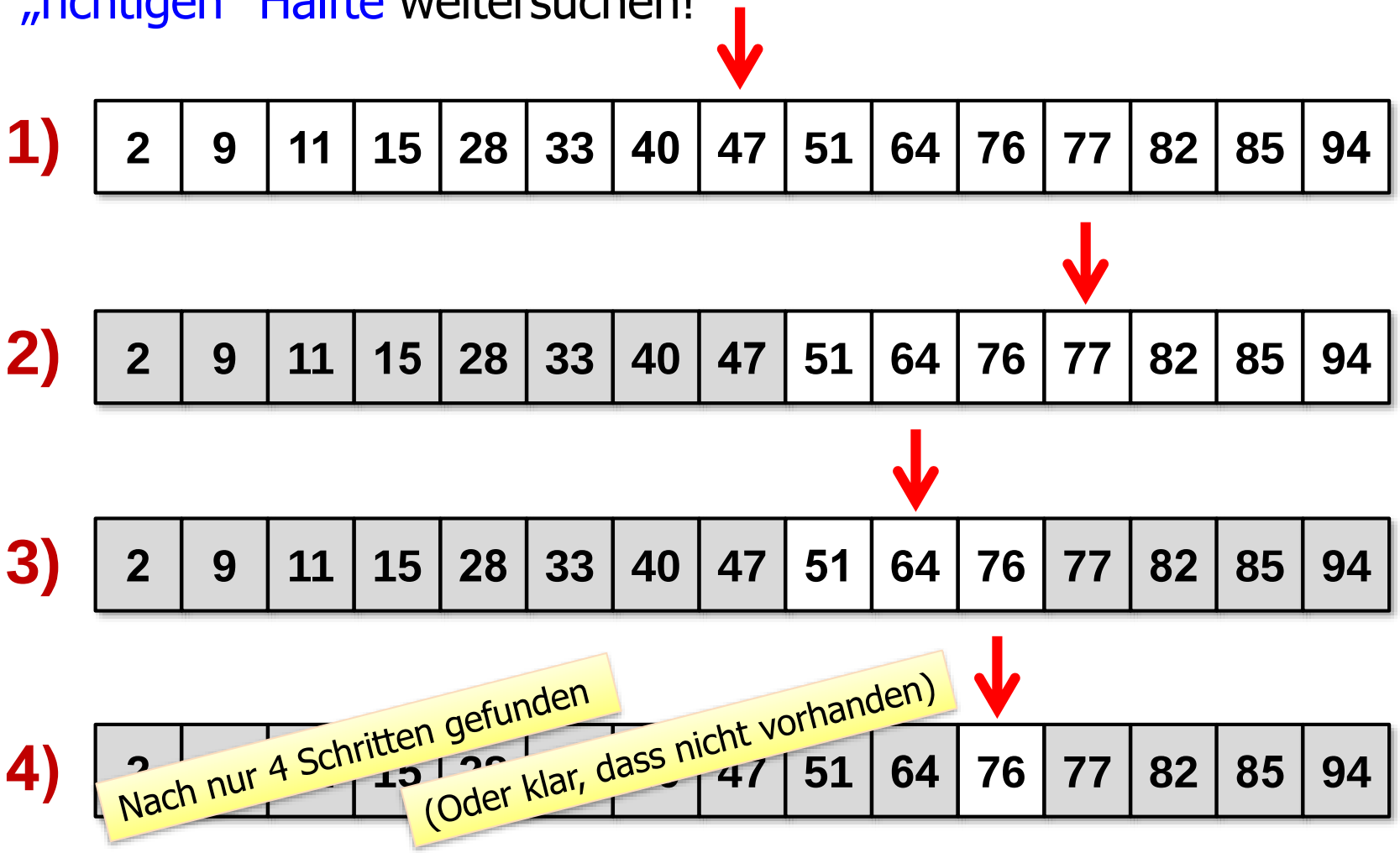
Die sogenannte **lineare Interpolationssuche** kann die Binärsuche noch deutlich beschleunigen. Die Idee ist, dass man nicht immer „blind" in die Mitte des Intervalls sticht, sondern aufgrund des gesuchten Schlüsselwertes und der angenommenen (gleichmässigen) Verteilung der Schlüsselwerte im Such-Datenbestand abschätzen kann, wo ungefähr (z.B. gegen Anfang oder aber ziemlich weit hinten etc.) sich ein gesuchtes Element höchstwahrscheinlich befindet (bzw. befinden müsste, wenn es vorhanden wäre). Man kann also ein Intervall mit einem bestimmten „Sicherheitsradius" um den vermuteten Aufenthaltsort bestimmen und dort iterativ weitersuchen. Optimierung und Analyse des Prinzips sind etwas vertrackt und man braucht Annahmen über die Verteilung. Daher begnügen wir uns in dieser Vorlesung mit der Grundidee und treiben es mit der Analyse der Interpolationssuche nicht auf die Spitze.

«Und wer da sucht, der findet» [Lk11,10]

- Aufgabe: Feststellen, ob ein Element in einem sortierten Array vorkommt
  - Und wenn ja, wo
- Lösung durch Einschachtelung (→ Intervallhalbierung):
  - Prüfen, in welcher Hälfte der gesuchte Wert liegen muss (wenn man nicht zufällig genau in der Mitte auf ihn gestossen ist!)
  - Verfahren dann iterativ (oder rekursiv) auf die „richtige" Hälfte anwenden → entweder linke oder rechte Grenze neu setzen
- Stoppkriterium?

Binärsuche auf sortierten Arrays

Beispiel: Ist die 76 vorhanden? → Jeweils in die Mitte stechen und in der „richtigen" Hälfte weitersuchen!

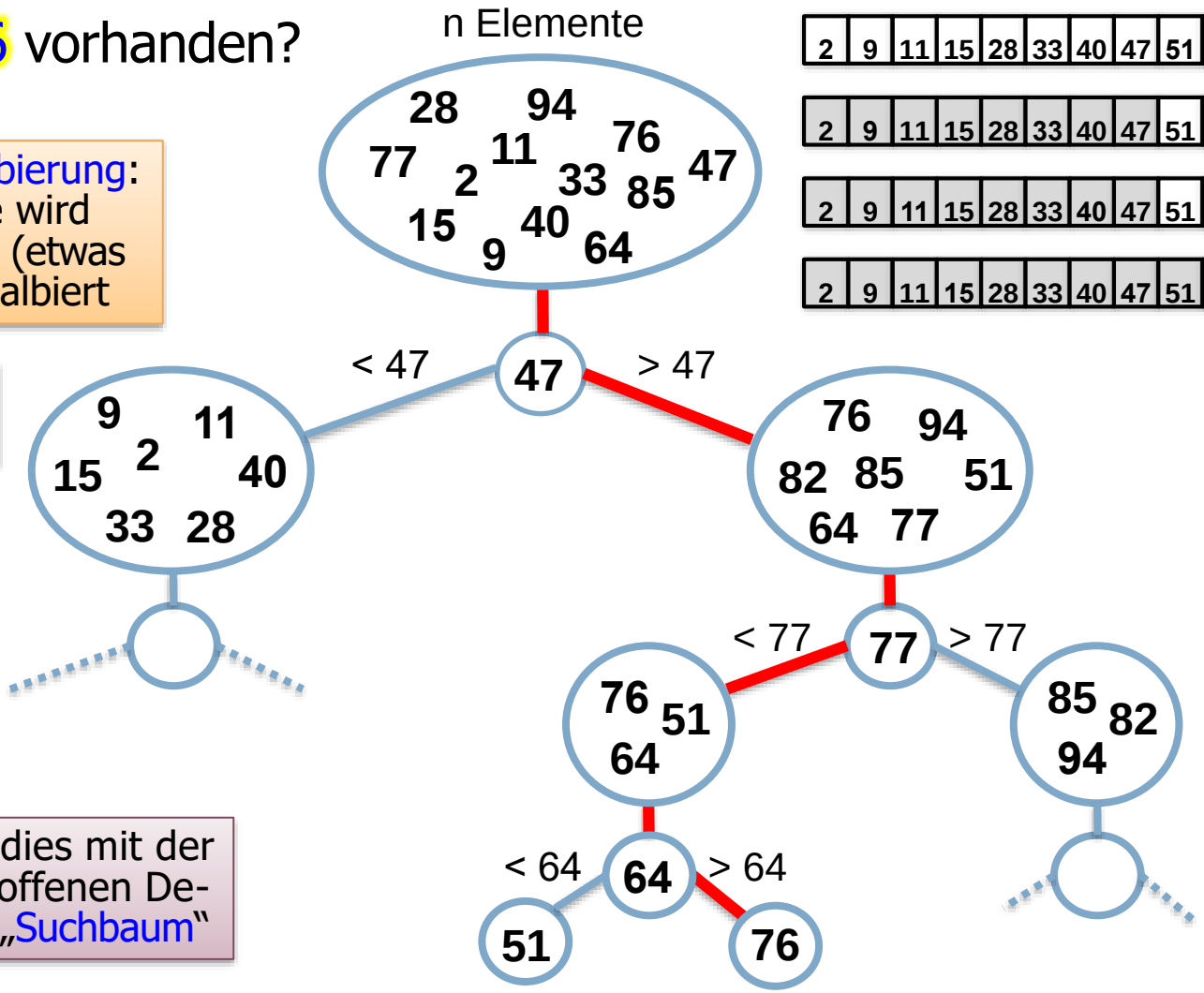


Binärsuche – Veranschaulichung durch Baum

Ist die 76 vorhanden?

Intervallhalbierung: Suchmenge wird schrittweise (etwas mehr als) halbiert

→ ca. $\log_2 n$ Schritte max.



2	9	11	15	28	33	40	47	51	64	76	77	82	85	94
2	9	11	15	28	33	40	47	51	64	76	77	82	85	94
2	9	11	15	28	33	40	47	51	64	76	77	82	85	94
2	9	11	15	28	33	40	47	51	64	76	77	82	85	94

Man vergl. dies mit der später getroffenen Definition von „Suchbaum“

Binärsuche in Java

```
// A sei hier ein globales int Array
int binSearch (int s) {
  int m;
  int li = 0; int re = A.length-1;

  while (re >= li) {
    m = (li+re)/2;
    if (s == A[m]) return m;
    if (s < A[m])
      re = m - 1;
    else
      li = m + 1;
  }

  return (-1);
}
```

Voraussetzung: A sei aufsteigend sortiert

Ist das auch OK, wenn
- li + re ungerade ist?
- oder wenn li = re?

Stoppkriterium:
- Wert gefunden → liefere Array-Index
- Restbereich „kollabiert" → liefere -1

Denkübungen:

1) Dürfen gleiche Elemente mehrfach vorkommen?

2) Was geschieht, wenn A doch nicht sortiert ist?
(Wie könnte dies effizient überprüft werden?)

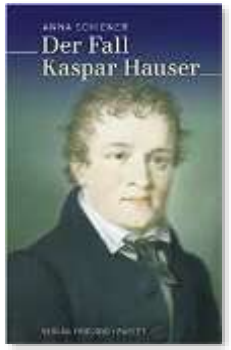
Binärsuche – Anwendungen

- Beispiel: Ist diese Telefonnummer schon vergeben?
- Oft hat man komplexere Datenstrukturen
  - Dann ist das Array nur bzgl. einer „Schlüsselkomponente“ sortiert
  - Beispiel: Wie lautet das Buch mit dieser ISBN-Nummer?

ISBN = 3150202562
Titel = Die Schatzinsel

ISBN = 3791722474
Titel = Kaspar Hauser

ISBN = 3360000978
Titel = Der Würger



- Sucheffizienz: Anzahl der Schritte ist logarithmisch zur Array-Länge
  - „Naive“ sequentielle Suche würde hingegen proportional lange dauern
- Beachte: Einfügen neuer Elemente ist allerdings „teuer“, da das Array sortiert bleiben muss (→ Platz schaffen durch Verschieben)

Binärsuche – Terminierung

Nochmal das Programm von eben:

```
// A sei hier ein globales int array
int binSearch (int s) {
  int m;
  int li = 0;
  int re = A.length-1;
  while (re >= li) {
    m = (li+re)/2;
    if (s == A[m]) return m;
    if (s < A[m])
      re = m - 1;
    else
      li = m + 1;
  }
  return (-1);
}
```

Terminiert die Schleife hier in jedem Fall?

Bei Schleifen ist nicht a priori klar, dass der Algorithmus nicht eventuell „stecken bleibt“ → Endlosschleife!

It is easier to write an incorrect program than to understand a correct one. [Alan Perlis, 1922 – 1990, Turing Award 1966]

Binärsuche – Terminierung (2)

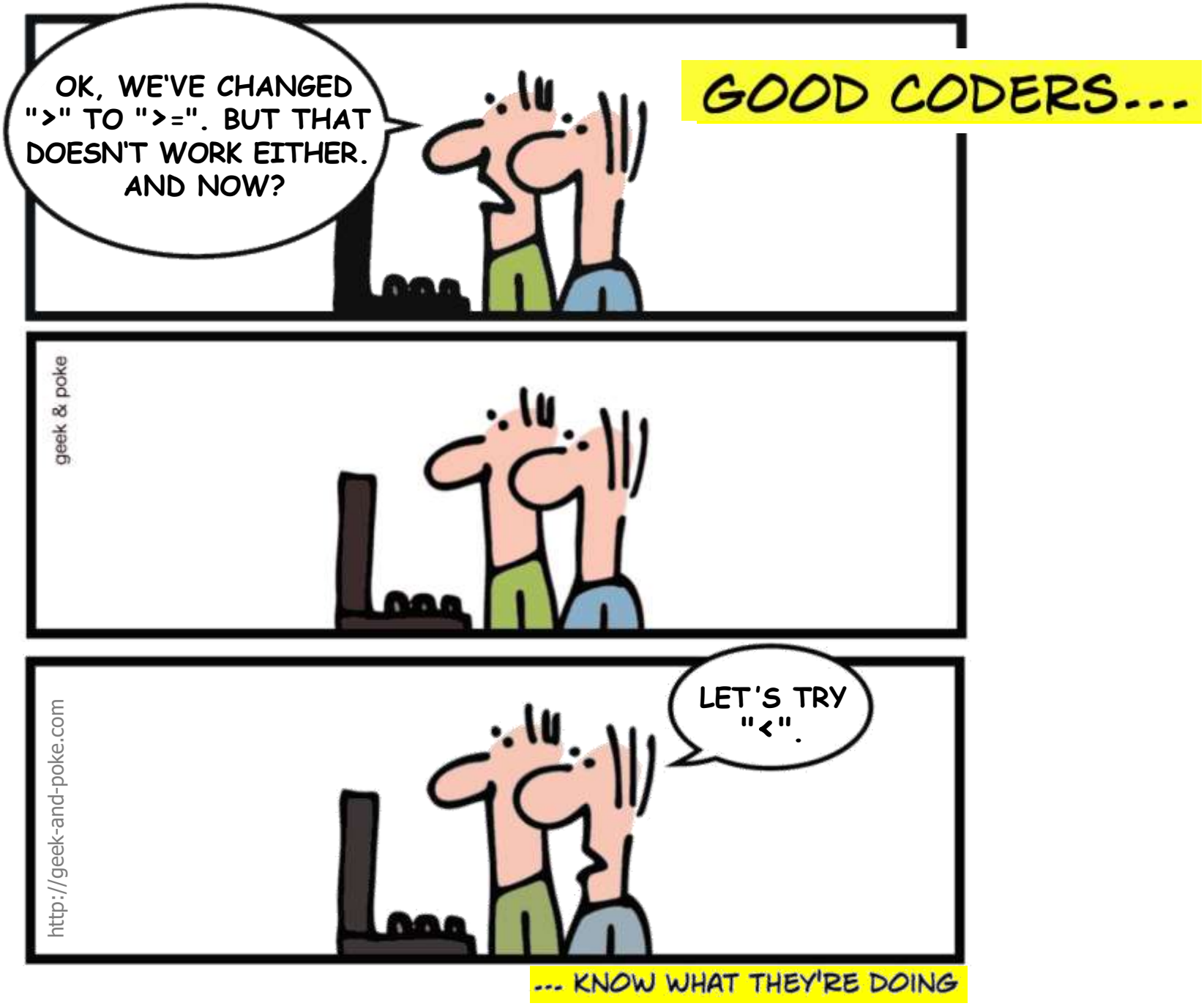
```
while (re >= li) {
  m = (li+re)/2
  if (s == A[m]) return m;
  if (s < A[m])
    re = m - 1;
  else
    li = m + 1;
}
```

„I was amazed: given ample time, only about ten percent of professional programmers were able to get this small program right." [Jon Bentley]

- Hätte man z.B. `... re = m; else li = m;`
  statt dessen, dann könnte folgendes geschehen:
  mit li = 4, re = 5 → m = (li+re)/2 = 4
  → (im Fall s > A[4]): li = 4, re = 5 → Endlosschleife!

- Wie können wir wirklich beweisen, dass so ein Problem bei unserer Lösung nicht vorhanden ist?
  - Korrektheitsbeweis (als Denkübung) *muss* bei `re = m-1; li = m+1`, aber *darf nicht* bei `re = m; li = m` funktionieren!

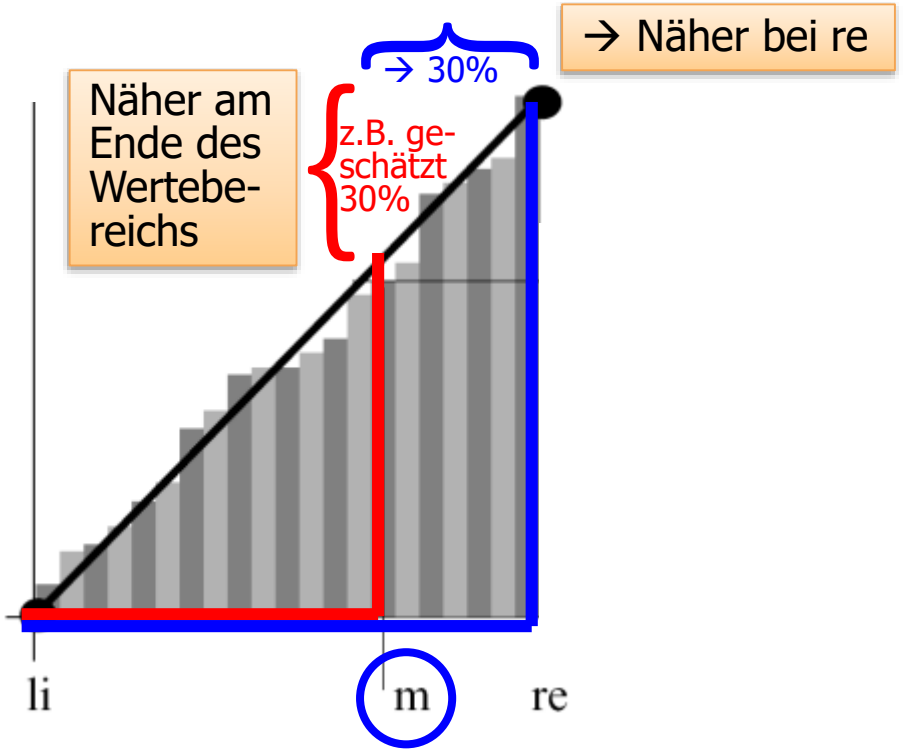
(Unvollständiges) Testen hilft nicht immer!


GOOD CODERS...
OK, WE'VE CHANGED ">" TO ">=". BUT THAT DOESN'T WORK EITHER. AND NOW?
geek & poke
LET'S TRY "<".
http://geek-and-poke.com
... KNOW WHAT THEY'RE DOING

Binärsuche – Optimierung?

- Kann man die Binärsuche noch verbessern?
- Idee: Beim Suchen in einem (physischen) Lexikon sticht man nicht blind in die Mitte, sondern...
  - → *lineare Interpolationssuche*

„***W**yss, Hans*“ steht z.B. relativ weit hinten



Lineare Interpolationssuche funktioniert wie Binärsuche, nur dass die „Mitte“ m jeweils besser ermittelt wird – als geschätzte Stelle, wo unter der Annahme einer Gleichverteilung das gesuchte Element sein sollte.

Lineare Interpolationssuche

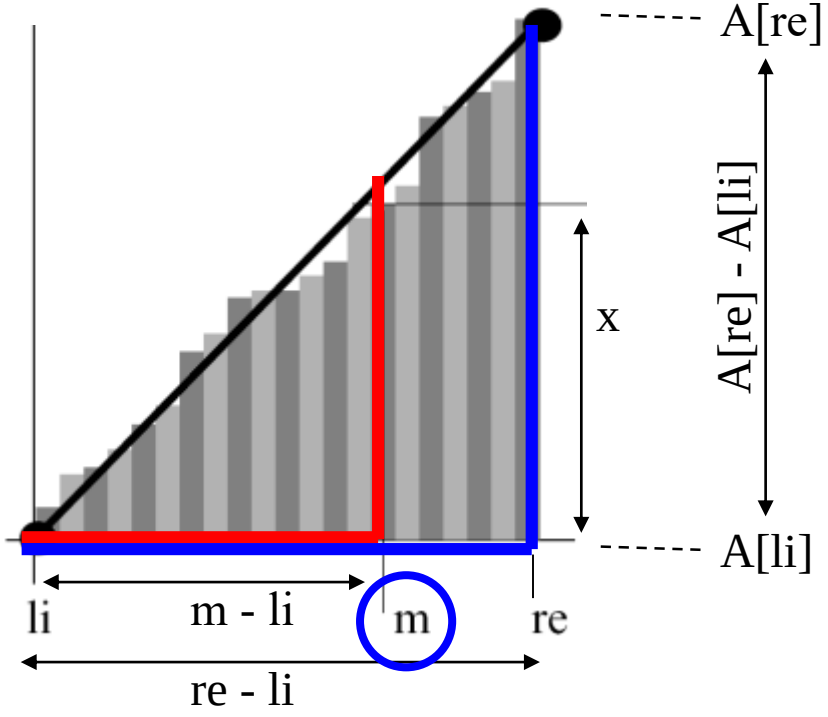
- Der mathematische Ansatz
  - → Strahlensatz aus der Geometrie

Setzt voraus, dass auf den Schlüsselwerten ein Mass definiert ist und sie einigermassen gleichförmig verteilt sind.

Ansatz: $\frac{x}{m - li} = \frac{A[re] - A[li]}{re - li}$

Gesucht ist m, alles andere ist bekannt

(Strahlensatz bzw. tan des Steigungswinkels)



Lineare Interpolationssuche (2)

mit $x = s - A[li]$ und Auflösung nach m

Ansatz: $\frac{x}{m-li} = \frac{A[re]-A[li]}{re-li}$

```
m = li + (s - A[li])*(re-li) / (A[re]-A[li]);
if (m < li) m = li;
if (m > re) m = re;
if (s == A[m]) return m;
if (s < A[m]) re = m-1;
         else li = m+1;
```

Aber würde man nicht durch 0 dividieren, wenn A[re] = A[li] ist?

Korrektur für den Fall, dass s ausserhalb des Bereichs liegt

Und würde man so nicht zu oft in der falschen („grösseren") Hälfte landen?

- Wie viel besser als die „normale" Binärsuche ist dies?
  - Typischerweise viel besser, wir analysieren das aber nicht weiter
- Findet z.B. Anwendung bei Zugriff auf externe Dateien
  - Wo jeder Zugriff relativ lange dauert und die etwas komplexere Berechnung relativ dazu vernachlässigbar ist
  - Dafür weitere Optimierungen, z.B. Index im Hauptspeicher halten

Binärsuche rekursiv und für generische Typen

„Comparable" sei ein Interface mit der abstrakten Methode „compareTo"

```
import ...

public class ... {
  public static int binSearch(Comparable x, Comparable [] a)
                    throws ItemNotFound {
    return bSearch(x, a, 0, a.length - 1); // Index liefern
  }

  // Lokal verborgene rekursive Methode:
  private static int bSearch
          (Comparable x, Comparable [] a, int low, int high)
          throws ItemNotFound {
    if (low > high) throw new ItemNotFound("Ham wa nich");
                                          // «Gits nüt»
    int mid = (low + high) / 2;
    if (a[mid].compareTo(x) < 0)
        return bSearch(x, a, mid+1, high);
    else if (a[mid].compareTo(x) > 0)
         return bSearch(x, a, low, mid-1);
    else return mid;  // Index des gesuchten Elements
  }
...
}
```

// ist x im array a?

Zwei Parameter

Vier Parameter

Aber lohnt sich Rekursion bei Binärsuche wirklich?

Testtreiber → Bonus-Slide

Binärsuche rekursiv – Testprogramm

```
// Test-Programm
public static void main(String [] args) {
  int SIZE = 8;
  Comparable [] a = new MyInteger [SIZE];

  for (int i = 0; i < SIZE; i++)
      a[i] = new MyInteger(i * 2);
  for (int i = 0; i < SIZE * 2; i++) {
    try {
      System.out.println("Found " + i + " at "
      + binSearch(new MyInteger(i), a));
    }
    catch(ItemNotFound e) {
      System.out.println(i + " not found");
    }
  }
}
```

Deklaration des Arrays „a“

Füllen von „a“ mit Werten

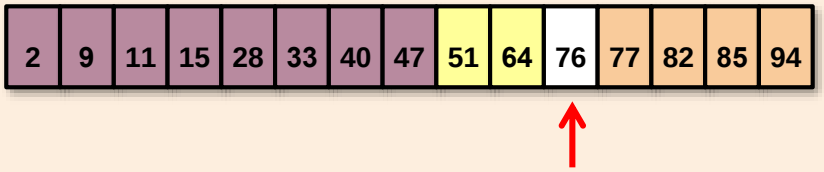
binSearch für Test-werte i aufrufen

Das Beispiel findet sich so ähnlich im Buch von Mark Weiss, Seite 308

- Benötigt wird noch die von „Comparable“ abgeleitete Klasse „MyInteger“, die die Methode „compareTo“ implementiert (→ als Übung)
- „ItemNotFound“ sei kanonisch von java.lang.Exception abgeleitet

Resümee des Kapitels

- Binärsuche auf sortierten Arrays
  - Fortgesetzte Intervallhalbierung
    → Suchaufwand ~ log n
  - Lineare Interpolationssuche



Vñ ich ſage euch auch/Bittet/ſo wirt euch gebenn/Sucht/ſo wer det yhr finden/ Klopfft an/ſo wirt euch auff than/Deñ wer do bittet/der nympt/vnd wer do ſucht/der findet/ vnd wer do an klopfft/ dem wirt auff than