

Informatik 2 mit BlueJ

Ein Kurs für die Stufe 12

von Ulrich Helmich

2. Auflage Mai 2009

Inhalt

Folge 12 - Suchalgorithmen	6
12.1 Lineare und binäre Suche	6
<i>12.1.1 Binäre Suche</i>	<i>6</i>
<i>12.1.2 Lineare Suche</i>	<i>6</i>
<i>12.1.3 Vergleich der beiden Suchverfahren</i>	<i>6</i>
12.2 Lineare Suche	7
<i>12.2.1 Analyseverfahren</i>	<i>7</i>
<i>12.2.2 Ergebnisse</i>	<i>7</i>
<i>12.2.3 Optimierungen der linearen Suche</i>	<i>9</i>
12.3 Binäre Suche	10
<i>12.3.1 Analyseverfahren</i>	<i>10</i>
<i>12.3.2 Ergebnisse</i>	<i>11</i>
<i>12.3.3 Genauere Analyse</i>	<i>11</i>
<i>12.3.4 Algorithmus der binären Suche</i>	<i>12</i>
Folge 13 - Quicksort	14
13.1 Grundprinzip des Quicksort	14
13.2 Der Quicksort im Detail	15
13.3 Optimierungen des Quicksort	20
Folge 14 - Abstrakte Datentypen	22
14.1 Datentypen und Datenstrukturen	22

14.2 Abstrakte Datentypen	23
14.3 Arbeitsweise eines Stacks	24
14.4 Implementierung einer Stack-Klasse	25
14.5 Die Klasse Stack im Zentralabitur NRW	30
14.6 Aufgaben mit der Klasse Stack	33
<i>14.6.1 Aufgabe: Kopieren eines Stacks</i>	<i>33</i>
<i>14.6.2 Aufgabe: Zusammenfügen von zwei Stacks</i>	<i>38</i>
14.7 Der ADT Queue	39
14.8 Der ADT Dictionary	41
14.9 Abstrakte Datentypen	42
14.10 Der ADT Stack - eine andere Sicht	44
Folge 15/16	46
Folge 17 - Referenzen	47
17.1 Was sind Referenzen?	47
17.2 Zeiger und Referenzen	48
17.3 Ein dynamischer Stack	49
17.4 Eine dynamische sortierte Liste	61
17.5 Eine dynamische Implementation der Klasse List	75
<i>17.5.1 Zielsetzung</i>	<i>75</i>
<i>17.5.2 Erweiterung der Klasse List zur NRW-List</i>	<i>76</i>

17.6	Expertenaufgaben	87
<i>17.6.1</i>	<i>Aufgabe: Vokabelliste</i>	<i>87</i>
<i>17.6.2</i>	<i>Aufgabe: Ein Super-Lexikon</i>	<i>90</i>
Folge 18	- Vererbung	92
18.1	Ein einfacher Fall der Vererbung	92
18.2	Heterogene Listen	104
18.3	Eine graphische heterogene Liste	115
Folge 19	- Bäume	117
19.1	Binärbäume - Allgemeines	117
19.2	Binäre Suchbäume	120
<i>19.2.1</i>	<i>Grundlegendes, Rekursivität</i>	<i>120</i>
<i>19.2.2</i>	<i>Vorteile binärer Suchbäume</i>	<i>121</i>
19.3	Binäre Suchbäume mit BlueJ	124
19.4	Eine rekursive Methode zum Anzeigen der Elemente	127
<i>19.4.1</i>	<i>Allgemeines</i>	<i>127</i>
<i>19.4.2</i>	<i>Arbeitsweise von showR()</i>	<i>128</i>
19.5	Eine insert()-Methode	132
<i>Fall 1</i>	<i>- Der Baum ist leer</i>	<i>132</i>
<i>Fall 2</i>	<i>- Der Baum enthält genau ein Element</i>	<i>132</i>
<i>Fall 3</i>	<i>- Der Baum enthält viele Elemente</i>	<i>132</i>
19.6	Eine rekursive insert-Methode	135
<i>Fall 1</i>	<i>- Der Baum ist leer</i>	<i>135</i>

Fall 2 - Der Baum ist nicht leer	135
Quelltext	135
Quelltext-Analyse	136
19.7 Der ADT „binärer Suchbaum,,	138
<i>Implementierung des ADT „binärer Suchbaum“ mithilfe eines Arrays</i>	139
19.8 Die Klasse Ordered Tree	141
19.9 Das Löschen von Elementen	143
<i>19.9.1 Suchen des zu löschenden Elementes</i>	143
<i>19.9.2 Das Löschen - Fall 1</i>	144
<i>19.9.3 Das Löschen - Fall 2</i>	146
<i>19.9.4 Das Löschen - Fall 3</i>	148
19.10 Ausgegliche Bäume	150
<i>19.10.1 Problemstellung</i>	150
<i>19.10.2 Einfügen in einen ausgeglichenen Baum</i>	152
<i>19.10.3 Vorteile von ausgeglichenen Bäumen</i>	154
19.11 BAYER-Bäume	156
<i>19.11.1 Grundidee</i>	156
<i>19.11.2 Eine mögliche Datenstruktur in Java</i>	158
Folge 20 - Containerklassen	160
20.1 Die Klasse ArrayList	160

Folge 12 - Suchalgorithmen

12.1 Lineare und binäre Suche

12.1.1 Binäre Suche

Wir wollen im Telefonbuch der Stadt Trier (ca. 100.000 Einwohner) nach der Nummer des Teilnehmers „Achim Lohmeier“ suchen. Wir wissen natürlich, dass ein Telefonbuch alphabetisch geordnet ist, und zwar nach den Hausnamen der Teilnehmer. Da der Name „Lohmeier“, mit einem „L“ anfängt und sich das „L“ im Alphabet im mittleren Teil befindet, schlagen wir das Telefonbuch auch ungefähr in der Mitte auf. Wir schauen auf den ersten Namen in der ersten Spalte der linken Seite und finden einen Herrn „Meierbrink“. Da haben wir also wohl etwas zu weit in der Mitte aufgeschlagen. Kein Problem, wir unterteilen den linken Teil des aufgeschlagenen Telefonbuchs wieder in zwei Hälften, eine etwas dickere linke Hälfte und eine etwas dünnere rechte Hälfte und schlagen das Buch dann dort auf. Jetzt finden wir den Namen „Kurrelmann“ in der ersten Spalte der linken Seite. Das „K“ kommt noch vor dem „L“, also müssen wir mit der Suche weiter rechts fortfahren. Nachdem wir noch zwei- oder dreimal nach dem gleichen Prinzip verfahren sind, finden wir schließlich die Seite mit dem Buchstaben L und können jetzt recht schnell unseren gewünschten Teilnehmer finden.

12.1.2 Lineare Suche

Wir wollen jetzt den Teilnehmer mit der Rufnummer 572356 im Ort Trier suchen. Da das Telefonbuch nicht nach den Telefonnummern sortiert ist, sondern nach den Hausnamen, bleibt uns nichts anderes übrig, als die Suche ganz oben in der ersten Spalte der ersten Seite zu beginnen. Wenn die Nummer nicht gefunden wurde, suchen wir in den restlichen Spalten der ersten Seite. Haben wir auch hier die Nummer nicht gefunden, machen wir auf der zweiten Seite weiter, dann auf der dritten und so weiter.

12.1.3 Vergleich der beiden Suchverfahren

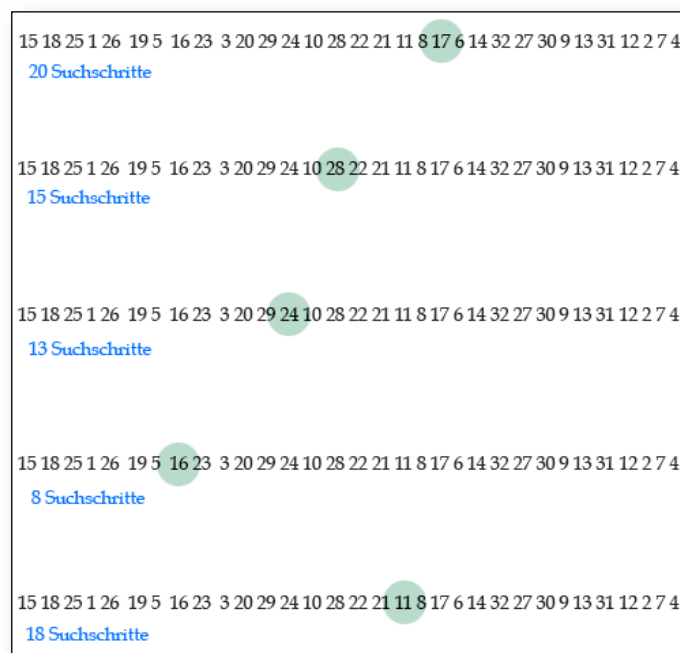
Die lineare Suche, auch als sequenzielle Suche bekannt, ist das einfachste Suchverfahren überhaupt - aber nicht das schnellste, wie wir gerade gesehen haben. Lineares Suchen hat zwei Vorteile: Erstens ist es recht einfach zu programmieren, und zweitens funktioniert es auch dann, wenn ein Array oder eine entsprechende andere Datenstruktur nicht sortiert ist. Der Nachteil der linearen Suche ist, dass es mitunter ziemlich lange dauern kann, bis das gesuchte Element gefunden ist. Steht die gesuchte Nummer ganz hinten im Telefonbuch, dauert die Suche wesentlich länger als wenn die Zahl vorne steht. Hier gibt es auch keine Optimierungsmöglichkeit, da die Telefonnummern nicht sortiert sind.

Die binäre Suche lässt sich nicht ganz so einfach beschreiben wie die lineare, und entsprechend komplexer ist auch der Algorithmus zur binären Suche. Der Vorteil der binären Suche ist die Geschwindigkeit, mit der Elemente gefunden werden können, falls sie überhaupt in der Datenstruktur vorkommen. Das werden wir uns gleich mit einem einfachen Beispiel klar machen. Der Nachteil der binären Suche: Sie kann nur auf bereits sortierte Datenmengen angewandt werden, und die Programmierung ist nicht so einfach wie die der linearen Suche.

12.2 Lineare Suche

12.2.1 Analyseverfahren

Bei der Analyse der beiden Suchverfahren wäre es recht umständlich, wenn wir ständig in dicken Telefonbüchern blättern müssten. Daher konstruieren wir uns ein sehr einfaches Modellsystem, das aus einem Array von 32 ganzen Zahlen besteht. Für die lineare Suche benutzen wir einen unsortierten Array, was den Telefonnummern in einem Telefonbuch entspricht, und für die binäre Suche verwenden wir einen sortierten Array, was den Hausnamen in einem Telefonbuch entspricht.



12-1 Lineare Suche in einem Array aus 32 int-Zahlen.

Die Abbildung 1 zeigt eine Zahlenliste mit absolut zufälligen Zahlen zwischen 1 und 32. Jede Zahl kommt genau ein Mal vor. Um die lineare Suche zu simulieren, suchen wir uns fünf beliebige Zahlen aus, zum Beispiel 17, 28, 24, 16 und 11. Dann wird nachgeschaut, an welcher Position diese Zahlen in der Liste vorkommen. Die 17 kommt an der Position 20 vor, also benötigt der lineare Algorithmus 20 Suchschritte, um diese Zahl zu finden. Ähnlich wird mit den anderen vier Zahlen verfahren.

12.2.2 Ergebnisse

Rechnet man die Suchschritte für alle fünf Zahlen zusammen und dividiert anschließend durch 5, so erhält man für unser Beispiel den Durchschnittswert 14,8.

Dieser Wert legt die Vermutung nahe, dass man bei N Zahlen wohl ungefähr $N/2$ Suchschritte zum Finden einer beliebigen Zahl benötigt. Der Suchaufwand kann also durch die Funktion $O(N) = N$ charakterisiert werden. Bei dieser Schreibweise ist es üblich, „Kleinigkeiten“ wie Faktoren, Kon-

stanten etc. zu vernachlässigen. Für einen linearen Algorithmus ist es egal, ob der Zeitaufwand N , $N/2$ oder $N/4$ ist, man sagt allgemein, der Zeitaufwand lässt sich durch eine Funktion f aus der Klasse $\mathbf{O}(N)$ charakterisieren. Wäre der Zeitaufwand dagegen quadratisch von N abhängig, wie dies z.B. beim Bubblesort der Fall ist, würde man sagen, der Zeitaufwand lässt sich durch eine Funktion $f \in \mathbf{O}(N^2)$ charakterisieren. Allgemein spricht man hier von der **O-Notation**, die häufig eingesetzt wird, um das **Zeitverhalten** von Algorithmen zu beschreiben.

Im günstigsten Fall (**best case**) befindet sich die gesuchte Zahl ganz vorne in der Liste und man braucht nur einen Suchschritt. Im ungünstigsten Fall (**worst case**) sucht man die letzte Zahl und benötigt N Suchschritte. Wenn die Zahl nicht in der Liste vorhanden ist, dann braucht man ebenfalls N Suchschritte.

Übung 12.1 (2 Punkte)

Berechnen Sie die durchschnittliche Suchzeit S für ganze Zahlen eines Arrays der Länge N , wenn W die Wahrscheinlichkeit dafür ist, dass die gesuchte Zahl im Array vorkommt ($0 < W \leq 1$).

Übung 12.2 (12 Punkte)

Erstellen Sie eine Klasse **Zahlen** mit einem Attribut **liste**; dieses Attribut soll ein Array aus 100 zufälligen int-Zahlen im Zahlenbereich zwischen 1 und 1000 sein. Jede Zufallszahl soll höchstens einmal im Array vorkommen (2 Punkte).

Statten Sie die Klasse mit Methoden zum Erzeugen und Anzeigen der Zahlen aus - achten Sie darauf, dass die Zahlen in ansprechender Weise angezeigt werden, nicht einfach alle untereinander oder nebeneinander, sondern zum Beispiel immer 16 oder 20 Zahlen in einer Reihe (2 Punkte).

Schreiben Sie dann eine Methode

```
public int sucheLinear(int suchzahl)
```

die einen linearen Suchalgorithmus realisiert; es soll nach der Zahl **suchzahl** gesucht werden. Wenn **suchzahl** gefunden wurde, gibt die Methode den Index der Fundstelle zurück, andernfalls den Wert -1 (2 Punkte).

Entwickeln Sie dann eine Methode, die **sucheLinear()** systematisch testet und für 10 zufällig gewählte Suchzahlen die durchschnittliche Suchzeit ermittelt. Von diesen 10 Zahlen sollen 7 in dem Array vorkommen, 3 jedoch nicht (5 Punkte).

Vergleichen Sie die so experimentell ermittelte durchschnittliche Suchzeit mit dem Ergebnis, das ihre Formel aus Übung 12.1 vorhersagen würde (1 Punkt).

12.2.3 Optimierungen der linearen Suche

Optimierungen des linearen Suchens sind dann möglich, wenn der Array bereits sortiert ist - die Frage ist allerdings, ob man dann nicht gleich zu einem binären Suchverfahren greifen sollte.

Bereits ohne Optimierung kann ein *sortierter* Array ungefähr *doppelt so schnell* durchsucht werden wie ein unsortierter. Man stelle sich vor, in dem unsortierten Array

12 - 18 - 4 - 7 - 13 - 6

soll die Zahl 9 gesucht werden. Um zu der Erkenntnis zu kommen, dass die 9 im Array *nicht* enthalten ist, benötigt der lineare Suchalgorithmus sechs Vergleiche. Betrachten wir nun den Fall, dass der Array sortiert ist:

4 - 6 - 7 - 12 - 13 - 18

Sobald der Suchalgorithmus die Zahl 12 „sieht“, steht zweifelsfrei fest, dass die 9 nicht im Array enthalten ist - nach nur vier Vergleichen.

Übung 12.3 (4 Punkte)

Der Array soll Zahlen zwischen 1 und 1000 enthalten und sortiert sein (einen Sortieralgorithmus haben Sie schnell eingebaut). Schreiben Sie nun eine Such-Methode, die folgende Optimierung enthält: Bei Suchzahlen kleiner als 500 beginnt die Suche links im Array, bei Suchzahlen ab 500 dagegen rechts - in diesem Fall wird der Array also von hinten ausgehend durchsucht.

Übung 12.4 (6 Punkte)

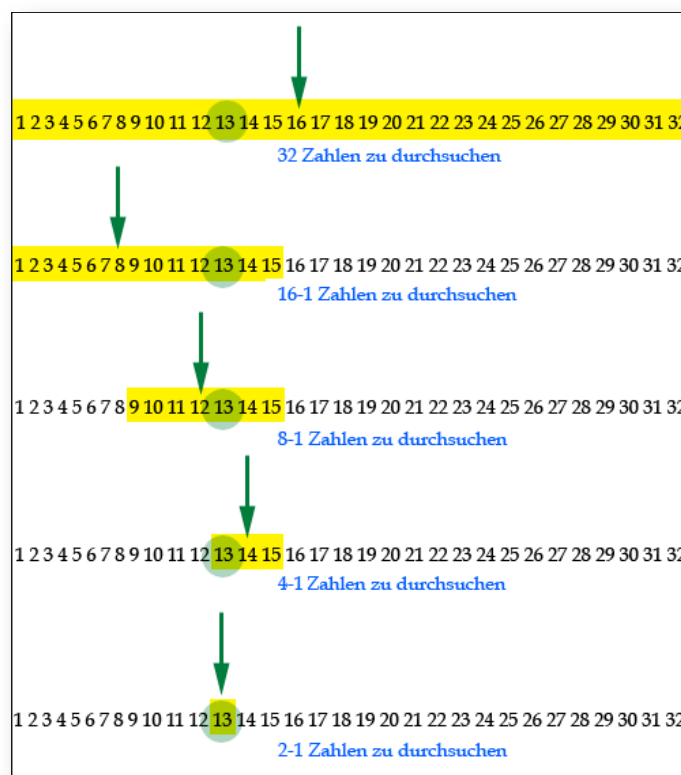
Das in 12.3 besprochene Verfahren kann man noch weiter optimieren. Bei 12.3 hatten wir *zwei* Startpositionen für die Suche, nämlich das Element mit dem Index 0 (ganz vorne) und das Element mit dem Index $N-1$ (ganz hinten), wobei N die Anzahl der Arrayelemente ist. Statt dieser zwei Startpositionen könnte man aber auch 10, 25 oder noch mehr Startpositionen in den Array einbauen und die Suche so enorm beschleunigen. Implementieren Sie eine derart optimierte Suche.

12.3 Binäre Suche

12.3.1 Analyseverfahren

Die binäre Suche funktioniert nur dann, wenn die zu durchsuchenden Objekte bereits sortiert sind. Für unsere Analyse erzeugen wir einen sortierten Array auf die denkbar einfachste Weise: Das erste Arrayelement erhält den Wert 1, das zweite Element den Wert 2 und so weiter.

Wir wollen die Zahl 13 suchen. Wir starten in der Mitte der 32 Zahlen. Bei 32 Elementen gibt es leider keine genaue Mitte, die Mitte würde zwischen den Elementen 16 und 17 liegen. Wir beginnen daher links von der Mitte mit der Zahl 16. Dann vergleichen wir die Suchzahl mit der mittleren Zahl des Suchbereichs, also der 16. Da die 13 kleiner als die 16 ist, setzen wir unsere Suche in dem Bereich links von der 16 fort.



12-2 Binäre Suche nach einer Zahl

Auf den linken Teilbereich wenden wir genau den gleichen Algorithmus an, der eben beschrieben wurde. Wir vergleichen die 13 mit der mittleren Zahl des linken Bereichs, der 8. Da 13 größer ist als 8, konzentrieren wir uns jetzt auf die rechte Hälfte der linken Hälfte.

Und so geht es weiter, bis wir nach nur fünf Suchschritten die 13 gefunden haben (siehe Abb. 12-3). Das ist ja fast unglaublich! Testen wir das Verfahren der binären Suche noch einmal an einer anderen Zahl, der 23.

12.3.2 Ergebnisse

Suchschritt 1: Die 23 ist größer als die 16, also machen wir rechts weiter.

Suchschritt 2: Die 23 ist kleiner als die 24, also machen wir links weiter.

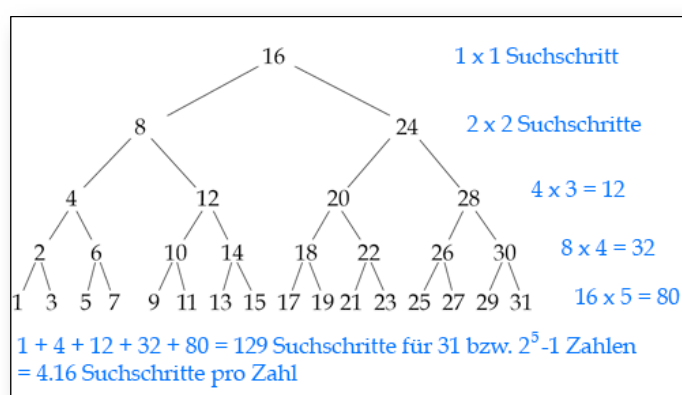
Suchschritt 3: Die 23 ist größer als die 20, also machen wir rechts weiter.

Suchschritt 4: Die 23 ist größer als die 22, also machen wir rechts weiter.

Suchschritt 5: in dem zuletzt übrig gebliebenen Intervall gibt es nur noch eine Zahl, nämlich die Suchzahl 23. Nach 5 Schritten sind wir also auch hier am Ende.

12.3.3 Genauere Analyse

Wir zeichnen 31 Zahlen nach folgendem Schema auf:



12-3 Ein binärer Suchbaum aus 31 Zahlen.

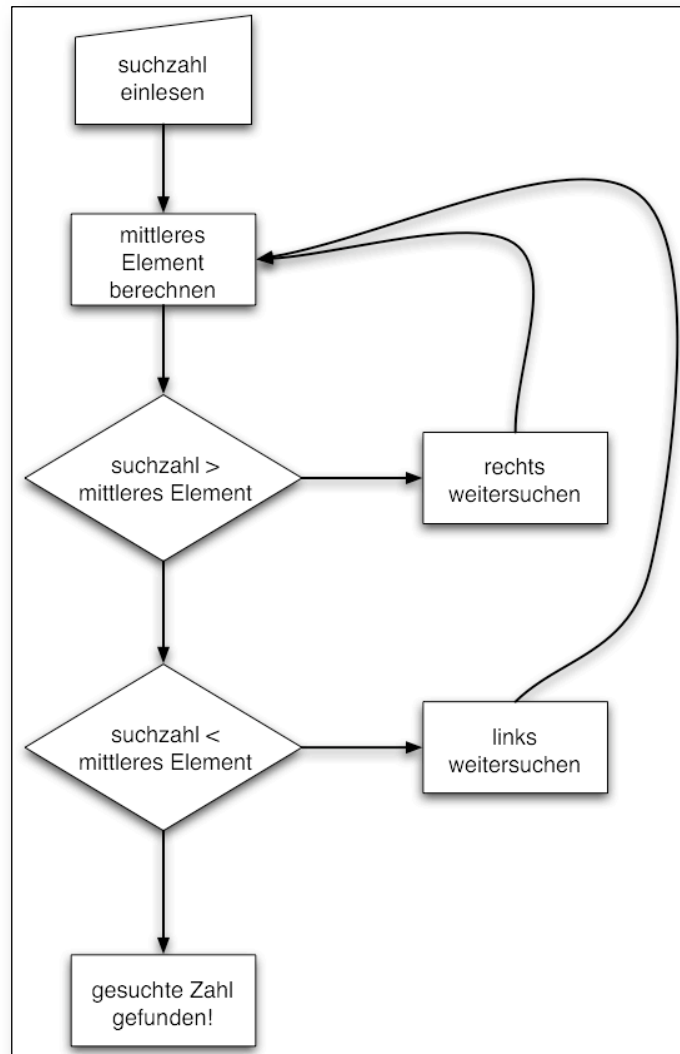
Wenn wir die 31 Zahlen in Form eines **binären Suchbaums** darstellen, wird der Vorteil der binären Suche sofort klar. Für das Suchen der Zahl 16 brauchen wir nur einen Suchschritt, für die Zahlen 8 und 24 zwei Suchschritte und so weiter. Wir kommen auf einen durchschnittlichen Wert von ca. 4 Suchschritten pro Zahl. Bei der linearen Suche brauchten wir für 31 Zahlen ca. 15 Suchschritte.

Vielleicht ist Ihnen schon aufgefallen, dass wir in der Abbildung 12-3 nur 31 Zahlen verwenden und nicht 32. Das liegt daran, dass man 31 Zahlen besser in einem Binärbaum unterbringen kann als 32 Zahlen. Mehr über Binärbäume erfahren Sie in der [Folge 19](#).

Übung 12.5 (1 Punkt)

- Wo müsste die Zahl 32 in dem Binärbaum in Abbildung 12-3 eingefügt werden?
- Wie groß wäre die durchschnittliche Anzahl der Suchschritte pro Zahl bei dem Binärbaum mit 32 Zahlen?

12.3.4 Algorithmus der binären Suche



12-4 Flussdiagramm zur binären Suche

Schritt 1

Die Mitte des Arrays wird berechnet. Seien `links` und `rechts` die Indices des ersten und des letzten Arrayelementes, so berechnet sich die Mitte nach folgender Gleichung:

$$\text{mitte} = (\text{links} + \text{rechts}) / 2$$

Da `mitte`, `links` und `rechts` int-Zahlen sind, findet eine ganzzahlige Division statt. Ein Array mit 32 Elementen hat folgende Daten: `links = 0`, `rechts = 31`. Daraus berechnet sich `mitte` mit $31/2 = 15$. Die Zahl mit dem Index 15 ist also jetzt das mittlere Element.

Hat der Array 33 Elemente, so erhält man mit `rechts = 32` die Arraymitte bei $32/2 = 16$. Das Gleiche gilt für einen Array mit 34 Elementen. Bei einem Array aus 35 oder 36 Elementen wäre 17 die Mitte und so weiter.

Schritt 2

Jetzt wird das mittlere Element mit der Suchzahl verglichen. Ist die Suchzahl *größer* als das mittlere Element, so muss *rechts* weiter gesucht werden. Ist die Suchzahl *kleiner* als das mittlere Element, so wird *links* weitergesucht. Trifft keine der beiden Bedingungen zu, so ist die mittlere Zahl die gesuchte Zahl, und der Algorithmus kann abbrechen. Die Abbildung 12.4 zeigt ein **Flussdiagramm** dieses Verfahrens.

Man kann diesen Algorithmus *rekursiv* programmieren oder in eine while-Schleife einbauen. Auf jeden Fall ist sicherzustellen, dass das mittlere Element bei jedem Durchgang neu berechnet wird. Eine mögliche rekursive Funktion könnte folgenden Funktionskopf haben:

```
private int sucheRekursiv(int suchzahl, int l, int r)
```

Dabei sind `suchzahl` die zu suchende Zahl und `l` und `r` die linke bzw. rechte Grenze des Arrays. Wird die Funktion zum ersten Mal aufgerufen, sind `l = 0` und `r = N-1`, wobei `N` die Zahl der Arrayelemente ist. Bei einem aus 32 Zahlen bestehenden Array würde also folgender Aufruf stattfinden:

```
ergebnis = sucheRekursiv(suchzahl, 0, 31);
```

wobei `ergebnis` eine int-Variable ist, die das Funktionsergebnis von `sucheRekursiv()` speichert. Die Frage ist jetzt, welches Ergebnis soll `sucheRekursiv()` eigentlich zurückgeben? Ziel unseres Vorgehens ist es ja, das *Zeitverhalten* eines binären Suchalgorithmus zu verstehen. Also wäre es sinnvoll, wenn das Funktionsergebnis die *Anzahl der zum Finden der Suchzahl erforderlichen Vergleiche* wäre. Bei einem Array mit 32 Zahlen würde der Rückgabewert von `sucheRekursiv()` also zwischen 1 (Suchzahl = Zahl genau in der Mitte) und 5 liegen. Falls die Zahl nicht im Array vorhanden ist, sollte als Ergebnis ebenfalls der Wert 5 zurückgegeben werden.

Das **Prinzip der Datenkapselung** verlangt, dass die Parameter einer Methode keine Rückschlüsse auf den intern eingesetzten Algorithmus zulassen. Würde man nun zwei Arraygrenzen als Parameter übergeben, so würde man dieses Prinzip verletzen. Also schreiben wir eine zweite Methode, diesmal eine öffentliche:

```
public int suchzeitBinaer(int suchzahl)
{
    return sucheRekursiv(suchzahl,0,31);
}
```

Diese öffentliche Methode *ummantelt* sozusagen die private rekursive Methode, um Informationen über eine mögliche Implementierung zu verbergen (**information hiding**).

Übung 12.6 (6 Punkte)

Ergänzen Sie die Klasse **Zahlen** um ein *rekursives* binäres Suchverfahren.

Testen Sie dann das binäre Suchverfahren mehrmals mit verschiedenen im Array enthaltenen und auch nicht enthaltenen Suchzahlen,

- ob es überhaupt funktioniert und
- welche Suchzeiten zum Finden der Suchzahlen benötigt werden.