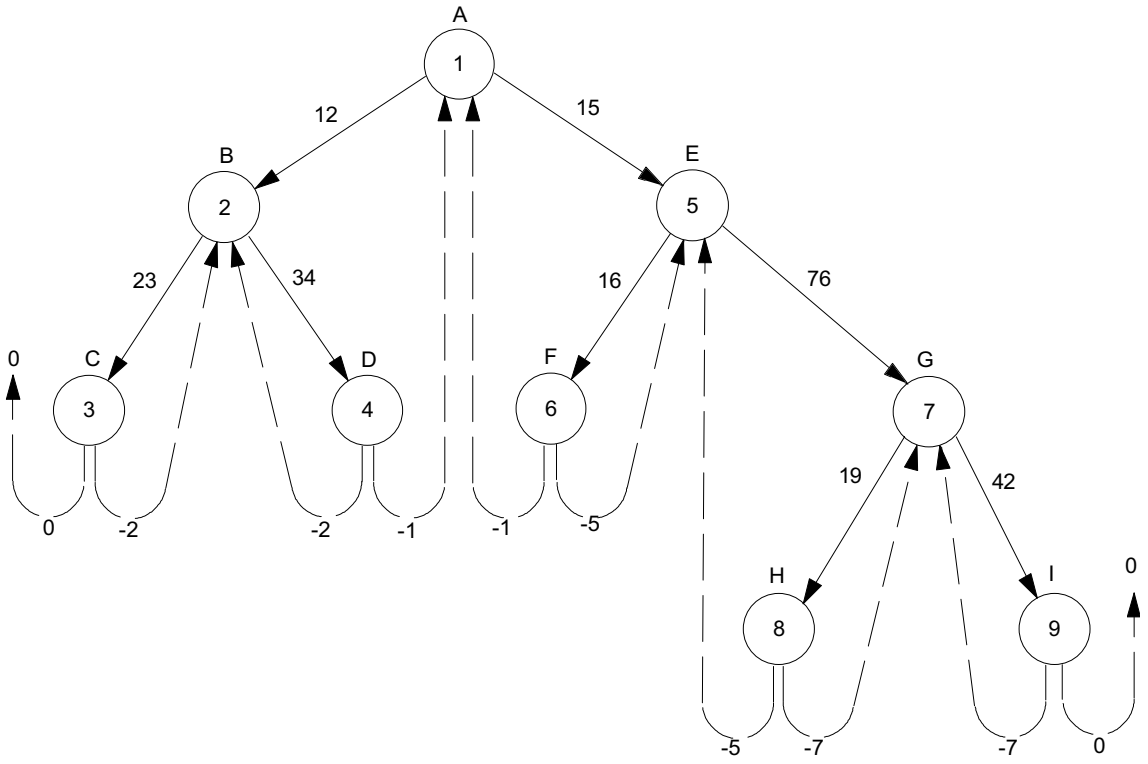
FernUniversität in Hagen



Algorithmen und Datenstrukturen

Kurseinheit 3:

Grundlegend

Ba

LESEPROBE
zum Kurs 814
Algorithmen und Datenstrukturen

Autor:
Prof. Dr. Hermann Gehring
unter Mitarbeit von:
Dr. Andreas Bortfeldt

I. Inhaltsübersicht

Kurseinheit 1: Algorithmen und Datenstrukturen

Kurseinheit 2: Algorithmen und Datenstrukturen

Kurseinheit 3: Algorithmen und Datenstrukturen

Kurseinheit 4: Algorithmen und Datenstrukturen

II. Leseprobe

1. Auszug aus Kurseinheit 3, Kapitel 4

4. Grundlegende Algorithmen

Im vorliegenden Kapitel werden einige Algorithmen vorgestellt, die – ähnlich wie die in Kap. 3 behandelten Operationen auf linearen Datenstrukturen – in unterschiedlichen betrieblichen Anwendungssystemen auftreten können. Aufgrund ihres anwendungsübergreifenden Charakters erscheint die Einstufung als "grundlegende" Algorithmen gerechtfertigt.

Suchalgorithmen, Sortieralgorithmen und rekursive Algorithmen

Zu diesen Algorithmen gehören u.a. Suchalgorithmen (siehe Kap. 4.1) und Sortieralgorithmen (siehe Kap. 4.2). Such- und Sortieralgorithmen eignen sich als spezielle Verfahren zur Lösung von klar abgrenzbaren Standardproblemen im Rahmen von betrieblichen Anwendungen, nämlich Such- und Sortierproblemen. Die in Kap. 4.3 behandelten rekursiven Algorithmen sind dagegen nicht auf eine spezifische Klasse von Problemen beschränkt. Charakteristisch für einen rekursiven Algorithmus ist vielmehr, daß er sich – direkt oder indirekt – selbst aufruft.

Zwischen Such- und Sortieralgorithmen sowie rekursiven Algorithmen besteht gegebenenfalls eine enge Beziehung. Sie rührt daher, daß sich Sortierprobleme auch auf rekursive Weise formulieren und lösen lassen (vgl. Kap. 4.3.3).

⋮

4.1 Suchalgorithmen

Suchoperationen können die Laufzeit von Anwendungssystemen in ungünstiger Weise beeinflussen, da bei der Suche von Datenobjekten unter Umständen zu sehr vielen oder gar allen Objekten eines gegebenen Bestandes zugegriffen werden muß. Es ist daher sinnvoll, sich grundsätzlich mit Suchverfahren auseinanderzusetzen und der Verfahrenseffizienz besondere Aufmerksamkeit zu widmen.

Organisationsformen für Datenbestände

Der Ablauf der Suche hängt von der Organisation des jeweiligen Datenbestandes und von der Art des Suchkriteriums ab. Hinsichtlich der Datenorganisation kann man u.a. folgende Fallunterscheidungen treffen:

fortlaufende Speicherung

(1) Suche in fortlaufend gespeicherten Datenbeständen, z.B. das Durchsuchen eines Arrays im Hauptspeicher oder die als ineffizient einzustufende Suche eines Satzes in einer auf einem Magnetband – etwa zu Sicherungszwecken – gespeicherten Datei.

verkettete Speicherung

(2) Suche in verketteten Datenbeständen, z.B. das Durchsuchen einer verketteten Liste im Hauptspeicher.

index-sequentielle Speicherung

(3) Suche in index-sequentiell organisierten Datenbeständen unter Nutzung einer Hilfsorganisation und gewöhnlich unter Einbezug von Externspeicherzugrif-

fen. Die angesprochene Hilfsorganisation umfaßt im Kern gewisse Hilfsdateien, die den raschen Zugriff auf extern abgelegte Daten gestatten.

gestreute Speicherung

(4) Suche in gestreut gespeicherten Datenbeständen unter Verwendung einer Adressierungsfunktion und – bei extern abgelegten Beständen – unter Einbezug von Externspeicherzugriffen.

Information Retrieval

(5) Information Retrieval, d.h. Suche in Dokumentenbeständen unter Nutzung von speziellen Hilfsorganisationen wie beispielsweise invertierten Dateien (engl. inverted files).

Suche auf linearen Datenstrukturen

In den Fällen (1) und (2) liegen lineare Datenstrukturen vor, auf denen die Suche ohne Nutzung einer Hilfsorganisation zu realisieren ist. Als Suchargument dient der Inhalt von Datenobjekten. In Frage kommen der Primärschlüssel oder andere Objektkomponenten und zwar als einzelnes oder aus mehreren Komponenten gebildetes Suchargument.

Suche unter Verwendung von Hilfsorganisationen und Adressierungsfunktionen

In den Fällen (3) und (4) wird die Suche unmittelbar durch die verwendete Hilfsorganisation bzw. Adressierungsfunktion beeinflußt. Dient der Primärschlüssel als Suchargument, so prägt die üblicherweise auf den Primärschlüssel bezogene Hilfsorganisation bzw. Adressierungsfunktion die Suche. Andernfalls kommen von der Hilfsorganisation bzw. Adressierungsfunktion unabhängige Suchverfahren in Betracht.

Verknüpfung von Suchargumenten

Für den Fall des Information Retrieval typisch sind Suchargumente, die aus mehreren, miteinander verknüpften Einzelkriterien bestehen. Solche gegebenenfalls recht komplexen Suchausdrücke lassen sich mittels spezieller Hilfsorganisationen auf effiziente Weise abarbeiten.

Auf Suchverfahren, die sich einer Hilfsorganisation oder einer Adressierungsfunktion bedienen, geht das vorliegende Kapitel nicht ein. Die Behandlung dieser Verfahren erfordert eine detaillierte Darstellung der jeweiligen Form der Datenspeicherung. Der Datenspeicherung ist ein eigenständiges Kapitel gewidmet (siehe Kapitel 6). Die Suche in index-sequentiell und gestreut gespeicherten Datenbeständen sowie das Information Retrieval werden daher in Kapitel 6 angesprochen.

binäre und sequentielle Suche auf linearen Datenstrukturen

Die hier behandelten Fälle (1) und (2) betreffen die Suche in linearen Datenbeständen, für die sich u.a. zwei Suchverfahren eignen, die sequentielle und die binäre Suche. Anders als die sequentielle Suche setzt die binäre Suche jedoch einen nach dem Suchkriterium sortierten Datenbestand voraus. Sind für die Objekte eines Bestandes Zugriffshäufigkeiten bekannt, so läßt sich die sequentielle Suche durch die Sortierung der Objekte nach der Zugriffshäufigkeit beschleunigen. Da die binäre Suche für verkettete Datenbestände nicht geeignet ist, ergeben sich die in Abb. 4.1 genannten Anwendungsmöglichkeiten der sequentiellen und der binären Suche.

Vergleich der sequentiellen und der binären Suche

Suchverfahren / Sortierung	Sequentielle Suche	Binäre Suche
unsortiert	sequentielle und verkettete Speicherung	—
nach Zugriffshäufigkeit sortiert	sequentielle und verkettete Speicherung	—
nach Suchkriterium sortiert	sequentielle und verkettete Speicherung	nur sequentielle Speicherung

Abb. 4.1. Sequentielle und binäre Suche in linearen Datenstrukturen.

Verwendung nur eines Sucharguments: Primärschlüssel

Unterstellt sei, daß lediglich ein Suchargument – nämlich der Primärschlüssel des zu durchsuchenden Datenbestandes – verwendet wird. Es genügt dann, die in Beispiel 4.1 formulierte Suchaufgabe zu betrachten. Sie eignet sich gleichermaßen zur Behandlung der sequentiellen und der binären Suche.

Beispiel zur Verwendung eines Sucharguments

Beispiel 4.1

Gegeben sei ein aus *n* Objekten bestehender Datenbestand. Der Bestand sei als Array organisiert. Die Typdefinition lautet:

TYPE DATENBESTAND = ARRAY [1..n] OF SATZ;

wobei SATZ eine Record-Struktur bezeichne. In dieser Struktur stelle die Komponente *key* den Primärschlüssel dar. Bezeichnet *searchkey* das Suchargument, so lautet die Suchaufgabe: Bestimme den Feldindex *pos* so, daß für die Variable *daten* vom Typ DATENBESTAND gilt:

daten[pos].key = searchkey

Die in Beispiel 4.1 genannten Typen und Variablen seien explizit und im Zusammenhang definiert:

Datendefinition mit numerischem Such-Schlüssel

```
DATA
    CONST n = 1500;
    TYPE
        INDEX =  [0..n + 1];
        SATZ =   RECORD
                 key : INTEGER;
                 beschreibung : ARRAY [1..12] OF CHAR;
                 ...;
                 END;
        DATENBESTAND = ARRAY [1..n] OF SATZ;
    VARIABLE
        daten : DATENBESTAND;
        searchkey : INTEGER;
```

Das Beispiel 4.1 und die dazugehörigen Datendefinitionen beinhalten zwei Einschränkungen:

- Als Suchargument wird nur der Primärschlüssel zugelassen.
- Es wird nur die sequentielle Speicherungsform betrachtet.

Diese Einschränkungen betreffen die binäre Suche, die laut Abb. 4.1 nur auf sequentiell und nach dem Primärschlüssel sortiert abgespeicherte Datenbestände anwendbar ist. Für die prinzipielle Vorgehensweise bei der sequentiellen Suche ist es dagegen unerheblich, ob der zu durchsuchende Datenbestand sequentiell oder verkettet gespeichert ist und ob das Suchkriterium nur aus einem einzelnen Suchargument oder aus einem komplexen Suchausdruck besteht. Insofern genügt es hier, lediglich die formulierte Suchaufgabe zu betrachten.

4.1.1 Sequentielle Suche

In Kapitel 1 wurde die sequentielle Suche bereits angesprochen (siehe Kap. 1.5, Beispiel 1.3). Die dort angegebene Suchaufgabe entspricht im wesentlichen der hier betrachteten Formulierung (siehe Beispiel 4.1).

Ist der zu durchsuchende Datenbestand nicht nach dem Primärschlüssel sortiert, so eignet sich folgende Prozedur zur sequentiellen Suche:

Prozedur zur sequentiellen Suche

```
PROCEDURE seqsuch( daten : DATENBESTAND;
                   searchkey : INTEGER;
                   VARIABLE found : BOOLEAN;
                   VARIABLE pos : INDEX);
BEGIN
    pos := 1;
    WHILE (pos < n + 1) AND (daten[pos].key ≠ searchkey) DO
        pos := pos + 1;
    ENDWHILE;
    IF (pos = n + 1) THEN
        found := FALSE;
    ELSE
        found := TRUE;
    ENDIF;
END seqsuch;
```

Beginnend beim ersten Satz wird mit Hilfe des laufenden Feldindex *pos* nacheinander auf die im Feld *daten* abgelegten Sätze zugegriffen bis schließlich der gesuchte Satz gefunden ist oder die Suche ohne Erfolg endet. Für die Anzahl der Durchläufe der WHILE-Schleife gilt:

Ablauf der Prozedur *seqsuch*

- Ist der erste Satz der gesuchte Satz, so wird die Schleife keinmal durchlaufen.
- Ist der letzte bzw. *n*-te Satz der gesuchte Satz, so wird die Schleife *n* - 1 mal durchlaufen.

- Ist der gesuchte Satz nicht in der Datei enthalten, so wird die Schleife nach *n* Durchläufen abgebrochen; der Index *pos* besitzt dann den Wert *n* + 1.

Im letztgenannten Fall zeigt die Statusvariable *found* die erfolglose Suche mit dem Wert FALSE an. In allen anderen Fällen war die Suche erfolgreich und *found* wird daher auf TRUE gesetzt.

Durch eine Vereinfachung des Bedingungsteils der WHILE-Schleife kann man die sequentielle Suche etwas beschleunigen. Allerdings muß dann der Datentyp DATENBESTAND um einen Platz erweitert werden:

Aufnahme eines zusätzlichen Datenobjekts als "Endmarke"

```
CONST n = 1500;
TYPE
    INDEX = [0 .. n + 1];
    DATENBESTAND = ARRAY [1 .. n + 1] OF SATZ;
```

Die Prozedur zur beschleunigten sequentiellen Suche lautet:

beschleunigte Fassung der Prozedur *seqsuch*

```
PROCEDURE bseqsuch (daten : DATENBESTAND;
                    searchkey : INTEGER;
                    VARIABLE found : BOOLEAN;
                    VARIABLE pos : INDEX);
BEGIN
    daten[n + 1].key := searchkey;
    pos := 1;
    WHILE (daten[pos].key ≠ searchkey) DO
        pos := pos + 1;
    ENDWHILE;
    IF (pos = n + 1) THEN
        found := FALSE;
    ELSE
        found := TRUE;
    ENDIF;
END bseqsuch;
```

Vereinfachung des Bedingungsteils der WHILE-Schleife

Die hinzugekommene (*n* + 1)-te Position des Feldes *daten* nimmt den Suchschlüsselwert auf. Ein Durchsuchen des Feldes *daten* führt also spätestens für den Index *pos* = *n* + 1 zum Lesen des gesuchten Schlüssels. Im Bedingungsteil der WHILE-Schleife ist daher die Einzelbedingung (*pos* < *n* + 1) überflüssig. Die Vereinfachung des Bedingungsteils bewirkt eine entsprechend geringere Suchzeit.

Indexwert zeigt den Erfolg der Suche an

Endet die Suche mit dem Indexwert *pos* = *n* + 1, so ist der gesuchte Satz nicht im Datenbestand enthalten. Andernfalls bezeichnet der Index *pos* mit einem Wert *pos* ∈ [1, *n*] die Feldkomponente, in welcher der gesuchte Satz abgelegt ist.

Suchwahrscheinlichkeiten

Sind die Sätze *daten[pos], pos* = 1,..., *n*, mit unterschiedlichen Wahrscheinlichkeiten Gegenstand der Suche, so läßt sich die Suche gegebenenfalls durch das Sortieren der Sätze nach diesen "Suchwahrscheinlichkeiten" beschleunigen. Be-

zeichne *p(pos), pos* = 1,..., *n*, die Wahrscheinlichkeit dafür, daß der unter dem Indexwert *pos* abgelegte Satz Gegenstand der Suche sei, wobei gilt:

$$\sum_{pos=1}^{n} p(pos) = 1. \tag{4.1}$$

Dann beträgt die Anzahl der Zugriffe zu Sätzen bei einer Suche im Mittel:

Anzahl der Zugriffe im Mittel

$$\bar{f}(n) = \sum_{pos=1}^{n} pos \cdot p(pos). \tag{4.2}$$

Werden alle Sätze mit gleicher Wahrscheinlichkeit gesucht, gilt also:

$$p(1) = p(2) = ... = p(n) = 1/n,$$

so erhält man für den Mittelwert der Rechenzeitfunktion *f(n)*:

Rechenzeitfunktion bei gleichen Suchwahrscheinlichkeiten

$$\bar{f}(n) = \sum_{pos=1}^{n} pos \cdot 1/n = 1/n \cdot \sum_{pos=1}^{n} pos = 1/n \cdot n \cdot (n+1)/2,$$

bzw.

$$\bar{f}(n) = (n+1)/2. \tag{4.3}$$

Dieses Ergebnis ist bereits aus dem in Kapitel 1.5 behandelten Beispiel 1.3 bekannt.

Sind im Falle ungleicher Suchwahrscheinlichkeiten die Sätze des Feldes *daten* nach fallenden Wahrscheinlichkeiten sortiert, gilt also:

$$p(1) \geq p(2) \geq ... \geq p(n),$$

so nimmt $\bar{f}(n)$ einen minimalen Wert an. Betrachtet werde ein hypothetischer Fall, der durch folgende Suchwahrscheinlichkeiten gekennzeichnet ist:

$$p(pos) = 1/(2^{pos}) \text{ für } pos = 1,..., n - 1, \tag{4.4}$$

und

$$p(pos) = 1/(2^{n-1}) \text{ für } pos = n. \tag{4.5}$$

Für den Mittelwert *f(n)* gilt dann (ohne Herleitung):

Rechenzeitfunktion bei fallenden Suchwahrscheinlichkeiten

$$\bar{f}(n) = 2 - 2^{1-n.} \tag{4.6}$$

Im Mittel wären also weniger als zwei Zugriffe erforderlich, um einen gesuchten Satz zu finden. Dieses Resultat erscheint plausibel, wenn man die nach den Be-

ziehungen (4.4) und (4.5) stark fallenden Suchwahrscheinlichkeiten berücksichtigt. Sie entwickeln sich wie folgt:

$p(1) = 1/2, p(2) = 1/4, p(3) = 1/8, p(4) = 1/16$ usw.

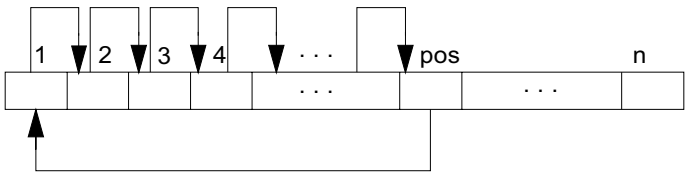
Die kumulierte Suchwahrscheinlichkeit für das erste und zweite Element beträgt demnach immerhin 0,75 bzw. 75 %.

In der Praxis werden numerische Werte für Suchwahrscheinlichkeiten kaum bekannt sein. Ist man sich jedoch sicher, daß die Suchwahrscheinlichkeiten der gesuchten Objekte erheblich differieren, so führt folgender pragmatischer Ansatz zu geringeren Suchzeiten:

Beispiel 4.2

Das Datenobjekt, das bei einer erfolgreichen sequentiellen Suche als zutreffend ermittelt wurde, zieht man auf die erste Feldposition vor. In der schematischen Darstellung bezeichnet *pos* dieses Objekt:

Einschwingphase beim Sortieren nach Zugriffshäufigkeit



Nach einer "Einschwingphase", in der ein aufwendiges Umspeichern der jeweils links von *pos* befindlichen Objekte in Kauf zu nehmen ist, stehen die am häufigsten benötigten Objekte am Anfang des Feldes.

Sequentielle Suche mit Vorziehen und Umspeichern

Verfolgt man den in Beispiel 4.2 genannten Ansatz, so kommt während einer Einschwingphase, deren Dauer von der Zugriffshäufigkeit zu dem betrachteten Datenbestand abhängt, die Prozedur *umspsuch* zur Anwendung. Nach dem Ende dieser Phase wird man beispielsweise die Prozedur *bseqsuch* anwenden. Handelt es sich bei dem durchsuchten Bestand um längerfristig gehaltene Daten, so ist eine Wiederholung der Einschwingphase in größeren Zeitabständen empfehlenswert.

4.1.2 Binäre Suche

setzt nach Suchkriterium sortierten Datenbestand voraus

sukzessive Zweiteilung des Datenbestands und Vergleich mit dem mittleren Element

Bekanntlich setzt die binäre Suche einen nach dem Suchkriterium sortierten Datenbestand voraus (vgl. hierzu Abb. 4.1). In der Regel liegt eine Sortierung nach steigenden Primärschlüsselwerten vor. Die Vorgehensweise ist etwa die folgende:

Der gegebene Suchschlüssel wird mit dem Schlüssel des Datenobjekts in der Mitte des Bestandes verglichen. Bei Gleichheit ist das Objekt das Gesuchte und die Suche wird beendet. Bei Ungleichheit wird festgestellt, ob der Suchschlüssel kleiner oder größer als der Schlüssel des Objekts in der Bestandsmitte ist. Abhängig davon liegt das gesuchte Objekt in der unteren oder oberen Bestandshälfte. Nun

wird auf das Objekt in der Mitte der zutreffenden Bestandshälfte zugegriffen und in analoger Weise wie zuvor für den gesamten Bestand verfahren. Die fortgesetzte Zweiteilung von Bestandsteilen führt schließlich zur Lokalisierung des gesuchten Objekts, vorausgesetzt, es ist im Bestand enthalten.

Das Prinzip der sukzessiven Zweiteilung von immer kleiner werdenden Bestandsteilen schlägt sich in dem Begriff "binäre" Suche nieder. Aus dem gleichen Grunde spricht man auch von bisektioneller Suche. Zur weiteren Verdeutlichung dieses Suchprinzips diene Abb. 4.2.

Suchschema der binären Suche

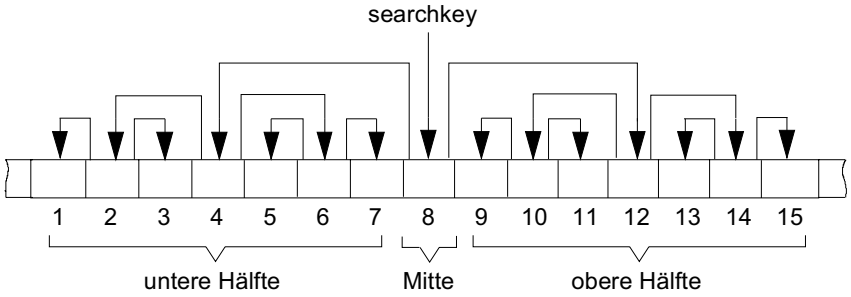


Abb. 4.2. Suchschema der binären Suche.

symmetrische Zugriffsstruktur

In Abb. 4.2 repräsentieren die Pfeile Zugriffe zu Datenobjekten. Auffällig ist die symmetrische Zugriffsstruktur. Sie zeigt, daß für das Lokalisieren eines Objekts minimal 1 Zugriff und maximal 4 Zugriffe erforderlich sind. Der Grund für die Ausgewogenheit der Zugriffsstruktur liegt im Umfang des Datenbestandes. Der Bestand umfaßt 2^4 - 1 bzw. 15 Sätze. Eine derart ausgewogene Zugriffsstruktur erhält man stets, wenn für die Anzahl der Sätze eines Bestandes gilt:

$$n = 2^r - 1,\ r = 2, 3, 4,... \tag{4.7}$$

Ist die Beziehung (4.7) erfüllt, so läßt sich die binäre Suche wie folgt als Prozedur formulieren:

Prozedur zur binären Suche

```
PROCEDURE binsuch(  daten : DATENBESTAND;
                    searchkey : INTEGER;
                    VARIABLE found : BOOLEAN;
                    VARIABLE pos : INDEX);
DATA
    VARIABLE ug, mitte, og : INDEX;     {Untergrenze, Mitte, Obergrenze}

BEGIN
    pos := 0;                                   {Zeiger initialisieren}
    ug  := 1;                                   {Untergrenze setzen}
    og  := n;                                   {Obergrenze setzen}
    found := FALSE;                             {Statusvariable initialisieren}
    WHILE (ug ≤ og) AND (NOT found) DO
        mitte := (ug + og)/2;
        IF (searchkey < daten[mitte].key) THEN
            og := mitte - 1;                    {Obergrenze korrigieren}
```

```
        ELSE
            IF (searchkey > daten[mitte].key) THEN
                ug := mitte + 1;            {Untergrenze korrigieren}
            ELSE
                found := TRUE;              {erfolgreiche Suche}
                pos := mitte;               {Satz anzeigen}
            ENDIF;
        ENDIF;
    ENDWHILE;
END binsuch;
```

Erläuterung zur Prozedur binsuch

Auf die Initialisierung der Zeigervariablen *pos* und der Statusvariablen *found* sowie das Setzen der Bestandsunter- und obergrenze *ug* und *og* folgt eine WHILE-Schleife, die den eigentlichen Suchprozeß beschreibt. Im Schleifenrumpf wird zunächst der Index *mitte* des Objekts in der Bestandsmitte berechnet. Es folgt eine IF-Anweisung, deren THEN-Zweig dann ausgeführt wird, wenn sich der gesuchte Satz in der unteren Bestandshälfte befindet. Die Ausführung besteht in der Korrektur der Obergrenze *og* derart, daß *ug* und *og* nunmehr die untere Bestandshälfte begrenzen; diese Hälfte würde dem weiteren Vorgehen – insbesondere der erneuten Berechnung des Index *mitte* – zugrunde liegen.

gesuchter Satz in unterer Bestandshälfte

oder

Befindet sich dagegen der gesuchte Satz nicht in der unteren Hälfte, so kommt der ELSE-Zweig der IF-Anweisung zur Ausführung. In diesem Zweig sind zwei weitere Fälle abzuhandeln:

- erstens, der gesuchte Satz befindet sich in der oberen Bestandshälfte, und
- zweitens, der Satz in der Bestandsmitte ist der gesuchte Satz.

in oberer Bestandshälfte oder in der Mitte

Dies geschieht mit Hilfe einer weiteren IF-Anweisung. Trifft der erstgenannte Fall zu, so wird die untere Bestandsgrenze entsprechend korrigiert. Andernfalls wird die Statusvariable *found* auf TRUE gesetzt und der Zeiger *pos* auf den gefundenen Satz gerichtet.

Ende bei erfolgloser Suche

Die Suche endet, falls der gesuchte Satz gefunden wurde – dann führt der Wert *found = TRUE* zum Abbruch der WHILE-Schleife – oder falls sich der gesuchte Satz nicht im Bestand befindet. Im letzteren Fall führt die Verletzung der Bedingung $ug \leq og$ zum Schleifenabbruch. Eine Verletzung dieser Bedingung tritt durch das Verschieben der Obergrenze *og* unter den Wert von *ug* oder durch das Verschieben der Untergrenze *ug* über den Wert von *og* auf.

Komplexität der binären Suche

Die Komplexität der binären Suche läßt sich wie folgt charakterisieren. Die Anzahl der Suchschritte (Vergleichsoperationen) beträgt im schlechtesten Fall:

$$f(n) = \lceil ld(n+1) \rceil, \tag{4.8}$$

und die mittlere Anzahl der Suchschritte ist für große *n* etwa:

$$\bar{f}(n) :\approx ld(n+1) - 1. \tag{4.9}$$

In den Beziehungen (4.8) und (4.9) steht ld für den Logarithmus zur Basis 2 (logarithmus dualis); $\lceil x \rceil$ bedeutet die kleinste ganze Zahl *y*, für die $y \geq x$ ist.

Vorteil der binären Suche

Im Vergleich zur sequentiellen Suche ist die binäre Suche ein sehr effizientes Verfahren. Beträgt die Anzahl der Sätze eines zu durchsuchenden Datenbestandes beispielsweise $n = 1000000$ (eine Million), so gilt für die mittlere Anzahl der Suchschritte:

- sequentielle Suche : $\bar{f}(n) = (n + 1)/2 \approx 500000$,
- binäre Suche : $\bar{f}(n) = ld\, n - 1 \approx 20$.

Die Anwendung der binären Suche setzt sortierten Datenbestand wahlfreien Zugriff und ein einfaches Suchkriterium voraus

Dem Vorteil der Effizienz der binären Suche, stehen mehrere Nachteile bzw. Einschränkungen gegenüber (vgl. auch die Ausführungen zu Beginn des Kap. 4.1):

- Eine zu durchsuchende Menge von Objekten muß nach dem Suchkriterium sortiert abgelegt sein. Dies führt bei anderen Operationen, so etwa bei dem Einfügen von Elementen, zu einem höheren Aufwand als bei unsortierter Speicherung.
- Auf die Objekte muß ein direkter Zugriff möglich sein, etwa mittels eines Feldindex als Selektor.
- Als Suchargument kommen keine miteinander verknüpften Einzelkriterien in Frage.

Berücksichtigt man die erwähnten Vor- und Nachteile, so kommt die binäre Suche vornehmlich dann in Betracht, wenn ein wenig variabler Datenbestand häufig zu durchsuchen ist.