Kapitel 3

Suchen

Das Suchen in Datenmengen ist eine der wichtigsten und grundlegendsten Operationen, die man mit Computern ausführen können möchte. Man denke an das Suchen nach einem Stichwort in einem Wörterbuch oder einer Enzyklopädie, die Suche nach einer Telefonnummer in einem Telefonverzeichnis, nach einem Namen in einer Symboltabelle, nach einer Kontonummer, Personalnummer, usw. Wir setzen in diesem Kapitel durchweg voraus, daß die Information, nach der wir suchen, durch einen Schlüssel eindeutig identifizierbar ist. Meistens nehmen wir der Einfachheit halber an, daß die Schlüssel positive ganze Zahlen sind, wie etwa Kontonummern, Personalnummern, Auftragsnummern. In der Praxis treten allerdings alphabetische Schlüssel, also Namen über einem endlichen Alphabet, ebenfalls häufig auf. Wir lassen bei der Diskussion verschiedener Suchverfahren die über den Schlüssel identifizierbare „eigentliche“ Information meistens unberücksichtigt. Als Argument für eine Suchoperation benutzen wir in der Regel also den Suchschlüssel k. Die Suche nach k in der Menge der gespeicherten Daten kann entweder *erfolgreich* enden bei einem Datum mit Schlüssel k oder aber *erfolglos*, falls es kein Datum mit Schlüssel k in der betrachteten Menge gibt. Im Falle einer erfolgreichen Suche nehmen wir natürlich an, daß wir über den Schlüssel auch Zugriff auf die „eigentliche“ Information haben und diese Information etwa lesen, ausgeben, verändern können.

Wir wollen in diesem Kapitel nur elementare Suchverfahren behandeln; das sind Verfahren, die, wie allgemeine Sortierverfahren, nur Vergleichsoperationen zwischen Schlüsseln ausführen. Arithmetische Operationen, die es erlauben, aus dem Suchschlüssel direkt die Speicheradresse zu berechnen, werden in diesem Kapitel ebensowenig behandelt wie besondere Datenstrukturen, die das Suchen und Wiederfinden gespeicherter Daten besonders unterstützen. Wir beschränken uns in diesem Kapitel im wesentlichen auf die Diskussion der wichtigsten elementaren Verfahren zum Suchen in linearen Listen, die sequentiell oder verkettet gespeichert sind. Arithmetische Verfahren zur direkten Bestimmung der Speicheradresse aus einem gegebenen Schlüssel werden ausführlich im Kapitel 4 über Hashing behandelt. Ebenso werden die wichtigsten die Suche unterstützenden Baumstrukturen in einem eigenen Kapitel 5 behandelt. Daß eine vorhandene Sortierung beim Suchen hilft, weiß jeder aus alltäglicher Erfahrung. Wir werden den möglichen Gewinn auch quantitativ abschätzen.

Wir haben etwa beim Sortieren durch Auswahl und bei Heapsort gesehen, daß manche Sortierverfahren auf der Suche nach dem kleinsten Schlüssel oder allgemeiner auf Verfahren zum Suchen nach dem i-kleinsten Schlüssel aufbauen. Diese Suchoperation wird im Unterschied zur Suche nach einem gegebenen Schlüssel üblicherweise als *Auswahl* (*Selection*) bezeichnet. Wir diskutieren das Auswahlproblem ebenfalls in diesem Kapitel und zwar im Abschnitt 3.1. Abschnitt 3.2 enthält die üblichen elementaren Suchverfahren für sequentiell gespeicherte Schlüssel. Im Abschnitt 3.3 diskutieren wir die wichtigsten Verfahren zur Selbstanordnung von linearen Listen. Das sind Verfahren, die nicht nur eine Suchoperation ausführen, sondern auch eventuell eine Umordnung der Liste vornehmen, um künftige Suchoperationen schneller durchführen zu können.

3.1 Das Auswahlproblem

Um das Element mit kleinstem oder größtem Schlüssel in einer Liste von N Elementen zu finden, genügt es, jedes Element der Liste einmal zu inspizieren. Sortieren ist nicht erforderlich. Sucht man das Element mit zweitkleinstem oder zweitgrößtem Schlüssel, kann man zunächst das Element mit kleinstem bzw. größtem Schlüssel bestimmen und aus der Liste entfernen und dann aus der Restliste von $N-1$ Elementen wiederum das Element mit dem kleinsten bzw. größten Schlüssel bestimmen. Auf analoge Weise fortfahrend kann man also das Element mit i-kleinstem Schlüssel, für jedes i mit $1 \leq i \leq N$, durch i-malige Bestimmung des Elements mit jeweils kleinstem Schlüssel aus Listen mit $N, N-1, \ldots, N-i+1$ Elementen in insgesamt $\Theta(i \cdot N)$ Schritten bestimmen. Für $i = N/2$ liefert dies insbesondere ein Verfahren zur Bestimmung des *mittleren* Elements aus einer Folge von N Schlüsseln, das Laufzeit $\Theta(N^2)$ hat. Das ist nicht sehr effizient, denn in Kapitel 2 haben wir gesehen, daß eine Folge von N Schlüsseln in $O(N \log N)$ Zeit sortiert werden kann. Da in einer sortierten Schlüsselfolge der i-kleinste Schlüssel in $O(N)$ Zeit bestimmt werden kann, läßt sich das mittlere Element einer Folge von N Schlüsseln offenbar mit Laufzeit $O(N \log N)$ bestimmen.

Gibt es Verfahren, die den i-kleinsten von N Schlüsseln und insbesondere den mittleren Schlüssel, den sogenannten *Median*, schneller zu bestimmen erlauben? Die zwei im Kapitel 2 behandelten Sortierverfahren Heapsort und Quicksort geben einen Hinweis darauf, wie eine Verbesserung des naiven Verfahrens erreicht werden könnte. Wir können versuchen, durch eine geeignete Datenstruktur die i-malige Bestimmung des jeweils kleinsten Schlüssels aus immer kleineren Listen zu beschleunigen; das führt zur Verwendung von Heaps analog zu Heapsort. Andererseits können wir eine Divide-and-conquer-Strategie analog zu Quicksort verfolgen, um den i-kleinsten von N Schlüsseln zu bestimmen. Wir diskutieren beide Möglichkeiten genauer.

Zur Bestimmung des i-kleinsten von N Schlüsseln können wir aus den gegebenen N Schlüsseln zunächst einen Heap herstellen. Anders als beim Verfahren Heapsort bauen wir aber einen sogenannten *Min-Heap* auf, also einen Heap, bei dem die Schlüssel der Väter stets *kleiner* (oder gleich) den Schlüsseln der Söhne sind. Das ist aber auch schon der einzige Unterschied zu den im Abschnitt 2.3 verwendeten Heaps. Man speichert also die Elemente in einem Array und kann das Array in $O(N)$ Schritten in einen Min-

Heap verwandeln. Das Element mit kleinstem Schlüssel steht an der Wurzel (d.h. an Position 1 im Array). Nun entfernt man i-mal nacheinander das jeweils kleinste Element aus dem Min-Heap, macht das letzte Element zur Wurzel und läßt es versickern (wie bei Heapsort). Das kostet jedesmal $O(\log N)$ Schritte. Auf diese Weise kann man das i-kleinste Element in insgesamt $O(N + i \log N)$ Schritten bestimmen. Insbesondere kann man so den Median in $O(N \log N)$ Schritten finden. Das ist schon besser als das naive Verfahren, aber keineswegs optimal. Denn die folgende Überlegung wird zeigen, daß man den i-kleinsten von N Schlüsseln stets in linearer Zeit, also in $O(N)$ Schritten bestimmen kann.

Um das Element mit i-kleinstem Schlüssel zu bestimmen, teilen wir die gegebene Folge von N Elementen wie bei Quicksort bezüglich eines geeignet gewählten Pivotelements in zwei Teilfolgen auf. Nach der Aufteilung wird aber (im Unterschied zu Quicksort) nur eine der durch Aufteilung entstandenen Teilfolgen weiter betrachtet.

Zum Aufteilen eines Bereichs $a[l], \ldots, a[r]$ von Elementen verwenden wir folgende Funktion:

```
function teile(l, r : integer; pivot : keytype) : integer;
{teilt den Bereich a[l],...,a[r] in zwei Gruppen:
 a[l],...,a[m−1] sind Elemente mit Schlüssel ≤ pivot,
 a[m],...,a[r] sind Elemente mit Schlüssel ≥ pivot;
 die Funktion liefert als Wert den Beginn m der zweiten Gruppe}
```

Eine mögliche Implementation für die Funktion *teile* kann man leicht aus der im Abschnitt 2.2 angegebenen Prozedur *quicksort* ablesen. Das gibt uns folgenden Algorithmus, um das Element mit i-kleinstem Schlüssel unter den Elementen $a[l], \ldots, a[r]$ zu bestimmen (anfangs ist $l = 1$ und $r = N$): Wir wählen ein Pivotelement v und teilen den Bereich mit Hilfe der Funktion *teile* auf. Falls $i \leq m - l$ ist, dann kommt das i-kleinste Element in der ersten durch Aufteilung entstandenen Gruppe vor. Falls $i > m - l$ ist, dann ist das Element mit i-kleinstem Schlüssel in $a[l], \ldots, a[r]$ das Element mit $i-(m-l)$-kleinstem Schlüssel in der zweiten durch Aufteilung entstandenen Gruppe $a[m], \ldots, a[r]$. Wenn schließlich $l = r$ (und $i = 1$) geworden ist, haben wir das gesuchte Element gefunden.

Man erhält das folgende naheliegende Programmgerüst für das Verfahren:

```
procedure auswahl (l, r, i : integer);
{liefert das Element mit i-kleinstem Schlüssel unter den Elementen
 a[l],...,a[r]; es wird r ≥ l und 1 ≤ i ≤ (r − l) + 1 vorausgesetzt}
var m, v : integer;
begin
  if r > l then {aufteilen}
    begin
      wähle Pivotelement v;
      m := teile(l, r, v);
      if (i ≤ m − l)
        then auswahl(l, m − 1, i)
        else auswahl(m, r, i − m + l)
    end
  else {r = l}
    {jetzt muß i = 1 sein, also ist a[l] das gesuchte Element}
end
```

Wenn wir das Pivotelement so ungünstig wählen, daß eine der durch Aufteilung entstehenden Folgen stets nur ein Element enthält und wir rekursiv jedesmal die andere Folge weiter betrachten müssen, benötigt dieses Verfahren zur Bestimmung des Elementes mit i-kleinstem Schlüssel in einer gegebenen Folge von N Elementen natürlich $\Omega(N^2)$ Schritte.

Man kann gegenüber diesem ungünstigsten möglichen Fall also nur dann etwas gewinnen, wenn es gelingt, das Pivotelement so zu wählen, daß man bei jedem Aufteilungsschritt mit Sicherheit einen bestimmten Bruchteil der noch zu betrachtenden Elemente ausschließen kann. Nehmen wir einfach einmal an, das Pivotelement könne stets so bestimmt werden, daß eine Aufteilung eines Bereichs von N Elementen nach diesem Pivotelement zwei Gruppen liefert, die jeweils $q \cdot N$ und $(1-q) \cdot N$ Elemente haben, mit einem festen Faktor q, $0 < q < 1$. Wir können ohne Einschränkung annehmen, daß q die größere der beiden Zahlen q und $(1-q)$ ist.

Nach der Aufteilung eines Bereichs der Länge N muß die Prozedur *auswahl* zur Bestimmung des i-kleinsten Elements dann höchstens für einen Bereich mit Länge $q \cdot N$ aufgerufen werden. Die Aufteilung selbst (mit Hilfe der Funktion *teile*) kann in $O(N)$ Schritten durchgeführt werden.

Bezeichnen wir mit $T(N)$ die Anzahl der Schritte, die erforderlich ist, um das Element mit i-kleinstem Schlüssel unter N Elementen mit Hilfe der Prozedur *auswahl* zu bestimmen, so gilt offenbar folgende Rekursionsgleichung:

$$
\begin{aligned}
T(N) &= T(q \cdot N) + c \cdot N, \text{ mit einer Konstanten } c \\
&\leq c \cdot N \cdot \sum_{i=0}^{\infty} q^i = c \cdot N \cdot \frac{1}{1-q} = O(N).
\end{aligned}
$$

Das Verfahren zur Bestimmung des i-kleinsten Elementes ist also in linearer Zeit ausführbar, wenn das Pivotelement stets richtig gewählt werden kann. Hier hilft die folgende von Blum u.a. vorgeschlagene *Median-of-median-Strategie* [20].

Um in einer Folge von N Elementen mit paarweise verschiedenen Schlüsseln das i-kleinste Element zu finden, benutze das folgende Verfahren *Auswahl*:

0. {Rekursionsabbruch}
   Falls $N <$ Konstante, berechne i-kleinstes Element direkt und Stop. Sonst:

1. Teile die N Elemente in $\lfloor \frac{N}{5} \rfloor$ Gruppen zu je fünf Elementen und höchstens eine Gruppe mit höchstens vier Elementen auf.

2. Sortiere jede dieser $\lceil \frac{N}{5} \rceil$ Gruppen (in konstanter Zeit) und bestimme in jeder Gruppe das mittlere Element. Es ist für alle Fünfergruppen eindeutig bestimmt und auch für die letzte Gruppe mit weniger als fünf Elementen, wenn diese Gruppe eine ungerade Zahl von Elementen hat. Hat die letzte Gruppe eine gerade Zahl von Elementen, so wähle das größere der beiden mittleren Elemente. Man erhält so insgesamt $\lceil \frac{N}{5} \rceil$ Mediane in Zeit $O(N)$.

$\leq v$

Mediane

v

$\geq v$

aufsteigend

sortierte

Fünfer-Gruppen

aufsteigend sortierte Mediane

Abbildung 3.1

3. Wende das Verfahren *Auswahl* rekursiv auf die $\lceil \frac{N}{5} \rceil$ Mediane an, um das mittlere Element v dieser Mediane zu finden. (Falls $\lceil \frac{N}{5} \rceil$ gerade ist, ist v das größere der beiden mittleren Elemente.) v heißt Median der Mediane. Abbildung 3.1 zeigt ein Beispiel für den Fall $N = 35$.

   Jetzt wählt man den Median der Mediane als Pivotelement für die Aufteilung aller Elemente und fährt wie bekannt fort:

4. Teile die N Elemente bezüglich v auf in zwei Gruppen: Gruppe 1 enthält die k Elemente, die kleiner als v sind und Gruppe 2 enthält die $N - k - 1$ Elemente, die größer als v sind. Die Aufteilung kann in Zeit $O(N)$ durchgeführt werden.

5. Falls $i \leq k$ ist, bestimmt man mit Hilfe des Verfahrens *Auswahl* rekursiv das i-kleinste Element unter den k Elementen der Gruppe 1. Ist $i > k + 1$, so bestimmt man mit Hilfe des Verfahrens *Auswahl* rekursiv das $i - (k + 1)$-te Element in der Gruppe 2. Falls $i = k + 1$ ist, hat man das i-te Element in der Ausgangsfolge gefunden (nämlich v).

Die Wahl von v als Median von Medianen sichert, daß mit Ausnahme der Gruppe, in der v selbst vorkommt, und der möglicherweise vorkommenden einzigen Gruppe mit weniger als fünf Elementen, jede Fünfergruppe mit mittlerem Element kleiner als v wenigstens drei Elemente enthält, die kleiner als v sind. Genauso enthält jede Fünfergruppe mit mittlerem Element größer als v wenigstens drei Elemente, die größer als v sind. (Abbildung 3.1 veranschaulicht die Situation für den Fall $N = 35$.) Also gibt es in der Ausgangsfolge wenigstens

$$3 \cdot (\lceil \frac{1}{2} \lceil \frac{N}{5} \rceil \rceil - 2) \geq \frac{3N}{10} - 6$$

Elemente, die kleiner als v sind, und ebenso viele Elemente, die größer als v sind.

Daraus folgt sofort, daß das Verfahren *Auswahl* im Schritt 5 in jedem Fall für höchstens $\lceil 7N/10+6 \rceil$ Elemente rekursiv aufgerufen werden muß. Die Median-of-median-Strategie sichert also, daß man nach der Aufteilung stets einen festen Bruchteil der noch zu betrachtenden Elemente ausschließen kann. Dennoch folgt die Linearität der Laufzeit des angegebenen Verfahrens zur Auswahl des i-kleinsten Elements noch nicht unmittelbar, da wir das Verfahren nicht nur im Schritt 5, sondern auch im Schritt 3 zur Bestimmung des Medians von $\lceil \frac{N}{5} \rceil$ Elementen rekursiv aufgerufen haben. Wir rufen das Verfahren also nicht nur einmal, sondern zweimal für einen jeweils unterschiedlichen Bruchteil der ursprünglich gegebenen Elemente auf. Daß die Laufzeit dennoch linear in N bleibt, sieht man folgendermaßen ein.

Sei wieder $T(N)$ die Anzahl der Schritte, die erforderlich ist, um das Element mit i-kleinstem Schlüssel unter N Elementen mit Hilfe des angegebenen Verfahrens zu finden. Dann liest man aus der Verfahrensbeschreibung sofort die folgende Rekursionsformel ab:

$$(*) \quad T(N) \le T\left(\left\lceil \frac{N}{5} \right\rceil\right) + T\left(\left\lceil \frac{7}{10}N + 6 \right\rceil\right) + a \cdot N,$$

mit einer Konstanten a.

Aus dieser Rekursionsformel läßt sich wie folgt eine obere Schranke für $T(N)$ ableiten. Wähle eine Konstante c so, daß $c \ge 80a$ und $c \ge T(N)/N$ für alle $N \le 91$ gilt. Wir zeigen durch Induktion, daß $T(N) \le cN$ gilt; dabei ist es ausreichend, den Induktionsschritt für $N > 91$ zu betrachten. In diesem Fall kann $T(N)$ abgeschätzt werden nach

$$\begin{aligned}
T(N) &\le c \cdot \left\lceil \frac{N}{5} \right\rceil + c \cdot \left\lceil \frac{7}{10}N + 6 \right\rceil + aN && \text{(nach Induktionsvoraussetzung, da } \lceil \tfrac{N}{5} \rceil \text{ und } \lceil \tfrac{7}{10}N + 6 \rceil \text{ für } N > 91 \text{ echt kleiner als } N \text{ sind)} \\
&\le c \cdot \frac{1}{5}N + c + c \cdot \frac{7}{10}N + 7c + aN \\
&= c \cdot \frac{9}{10}N + 8c + aN \\
&\le c \cdot \frac{9}{10}N + 8c + \frac{1}{80}cN && \text{(nach Wahl von } c\text{)} \\
&= c \cdot \left(\frac{73}{80}N + 8 \right) \\
&\le c \cdot N && \text{(wegen } N > 91\text{)}
\end{aligned}$$

Wir halten fest:

Satz 3.1 *Das i-te Element in einer Folge von N Elementen kann in höchstens $O(N)$ Schritten gefunden werden.*

Dieser Satz ist von erheblichem prinzipiellem Interesse. Das zu seinem Beweis von uns angegebene *Auswahl*-Verfahren ist jedoch kaum von praktischem Wert, weil viele Sonderfälle berücksichtigt werden müssen, die das Verfahren für kleine N kompliziert machen; die asymptotisch geringe Laufzeit macht sich erst für sehr große N bemerkbar.

3.2 Suchen in sequentiell gespeicherten linearen Listen

Wir nehmen an, daß die Elemente der zu durchsuchenden Liste Komponenten eines wie folgt vereinbarten Arrays sind:

$$\textbf{var } a: \textbf{ array } [0\,..\,maxN] \textbf{ of } item;$$

Die gegebenen N Elemente sollen an den Positionen $1,\ldots,N$ stehen, $N \leq maxN$; jedes Element $a[i]$ hat eine Schlüsselkomponente $a[i].key$.

3.2.1 Sequentielle Suche

Das einfachste Suchverfahren, das keinerlei weitere Voraussetzungen verlangt, ist die *sequentielle* oder *lineare* Suche. Wird ein Element mit gegebenem Schlüssel k gesucht, so durchlaufen wir alle Elemente des Arrays von vorn nach hinten oder umgekehrt und vergleichen den Schlüssel jedes Elements mit dem Suchschüssel. Die Suche kann erfolgreich abgeschlossen werden, sobald ein Element mit diesem Schlüssel k gefunden wurde. Um nicht immer prüfen zu müssen, ob bereits alle Listenelemente inspiziert wurden, verwendet man üblicherweise einen Stopper an Position 0, der dafür sorgt, daß eine am Listenende beginnende Suche auf jeden Fall erfolgreich endet. Das führt zur folgenden programmtechnischen Realisierung des Verfahrens:

```
procedure sequentialsearch (k : integer);
{durchsucht a[1],...,a[N] nach Element mit Schlüssel k}
var i : integer;
begin
  a[0].key := k; {Stopper}
  i := N + 1;
  repeat
    i := i − 1
  until a[i].key = k;
  if i ≠ 0
    then {a[i] ist gesuchtes Element}
    else {es gibt kein Element mit Schlüssel k}
end {sequentialsearch}
```

Es ist offensichtlich, daß das Verfahren im schlechtesten Fall $N+1$ Schlüsselvergleiche für eine erfolglose Suche benötigt. Wenn man annimmt, daß jede Anordnung der N Schlüssel gleichwahrscheinlich ist, wird man erwarten können, daß eine erfolgreiche Suche im Mittel

$$\frac{1}{N}\sum_{i=1}^{N} i = \frac{N+1}{2}$$

Schlüsselvergleiche ausführt.

Natürlich könnte man dieses Suchverfahren leicht auch für verkettet gespeicherte lineare Listen entsprechend implementieren. Das Verfahren macht nämlich von der Möglichkeit des direkten Zugriffs auf ein Element über seine Position innerhalb der Liste keinen Gebrauch. Das ist bei allen folgenden Verfahren anders.

Darüberhinaus setzen wir für den Rest des Abschnitts 3.2 voraus, daß die Listenelemente nach aufsteigenden Schlüsselwerten sortiert vorliegen. Es gilt also

$$a[1].key \leq a[2].key \leq \ldots \leq a[N].key.$$

3.2.2 Binäre Suche

Das *binäre Suchen* folgt der Divide-and-conquer-Strategie und kann am einfachsten rekursiv beschrieben werden:

Verfahren *binäres_Suchen* $(L : Liste; k : Schlüssel)$;
{*sucht in der Liste L mit aufsteigend sortierten Schlüsseln nach Element mit Schlüssel k*}

1. *Falls L leer ist, endet die Suche erfolglos; sonst betrachte das Element* $a[m]$ *an der mittleren Position m in L.*
2. *Falls* $k < a[m].key$*, durchsuche die linke Teilliste* $a[1], \ldots, a[m-1]$ *nach demselben Verfahren.*
3. *Falls* $k > a[m].key$*, durchsuche die rechte Teilliste* $a[m+1], \ldots, a[N]$ *nach demselben Verfahren.*
4. *Sonst ist* $k = a[m].key$ *und das gesuchte Element gefunden.*

Zur programmtechnischen Realisierung dieses Verfahrens ist es bequem, die Grenzen des zu durchsuchenden Bereichs explizit als Parameter einer rekursiven Prozedur mitzuführen.

```
procedure binsearch (l, r, k : integer);
{durchsucht a[l],...,a[r] nach einem Element mit Schlüssel k}
var m : integer;
begin
  m := (l + r) div 2;
  if l > r
    then {Liste leer, Suche endet erfolglos}
    else
      begin
        if k < a[m].key
          then binsearch(l, m - 1, k)
          else if k > a[m].key
                 then binsearch(m + 1, r, k)
                 else {a[m].key = k; Suche endet erfolgreich}
      end
end
```

Ein Aufruf im Hauptprogramm der Form *binsearch* $(1,N,k)$ liefert dann das gewünschte Ergebnis.

Die angegebene programmtechnische Realisierung des binären Suchens ist natürlich nur eine von vielen Möglichkeiten. Wir geben als zweite Variante noch eine iterative Version an.

```
function binsearch (k : integer) : integer;
{liefert den Index eines Elementes mit Schlüssel k im Bereich a[1],...,
 a[N], falls es ein Element mit diesem Schlüssel gibt, und 0 sonst}
var m, l, r : integer;
begin
  l := 1; r := N;
  repeat
    m := (l + r) div 2;
    if k < a[m].key
      then r := m − 1
      else l := m + 1
  until (k = a[m].key) or (l > r);
  if k = a[m].key
    then binsearch := m
    else binsearch := 0
end
```

Weil wir nach jedem Vergleich des Suchschlüssels k mit dem Schlüssel des mittleren Elementes des zu durchsuchenden Bereichs die Hälfte der noch zu betrachtenden Elemente ausschließen können, folgt unmittelbar, daß bei binärer Suche für erfolgreiche und erfolglose Suche in einem Array mit N Elementen niemals mehr als $\lceil \log_2(N+1) \rceil$ Schlüssel miteinander verglichen werden.

Zur Abschätzung des mittleren Suchaufwands belastet man üblicherweise das Inspizieren eines Schlüssels und die gegebenenfalls notwendige Entscheidung, in der linken oder rechten Hälfte weiterzusuchen, mit den Kosten 1. Die zum Wiederfinden eines Elements erforderlichen Kosten sind dann gleich der Zahl der ausgeführten Schlüsselvergleiche nur unter der Annahme, daß das Verfahren binäre Suche in einer Programmiersprache implementiert wird, die einen Vergleichsoperator mit drei möglichen Ausgängen besitzt. Man nimmt also an, daß man in einem Schritt feststellen kann, ob ein gesuchter Schlüssel gleich, kleiner oder größer als ein inspizierter Schlüssel ist. Man beachte, daß beispielsweise die iterative Pascal Version des Verfahrens binäre Suche jeweils zwei Schlüsselvergleiche für diese Feststellung benötigt.

Um den mittleren Suchaufwand des Verfahrens binäre Suche abschätzen zu können, nehmen wir an, daß $N = 2^n - 1$ ist, für passendes n. Dann erfordert das Wiederfinden des Elementes an der mittleren Position genau eine Kosteneinheit, das Wiederfinden der Elemente an der mittleren Position in der jeweils linken und rechten Hälfte genau zwei Kosteneinheiten, usw. Es werden also genau $(i+1)$ Kosteneinheiten benötigt, um eins von 2^i Elementen wiederzufinden, $i = 0,\dots,n-1$. Für $n = 3$, also $N = 2^3 - 1 = 7$, kann man diesen Zusammenhang durch Abbildung 3.2 veranschaulichen.

Damit ergibt sich für den mittleren Suchaufwand des binären Suchens:

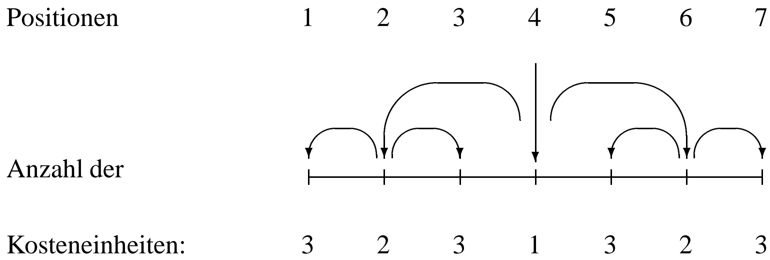

Abbildung 3.2

$$
\begin{aligned}
C_{mit}(N) &= \frac{1}{N}(\text{Gesamtkosten}) \\
&= \frac{1}{N}\sum_{i=0}^{n-1}(i+1)\cdot 2^i \\
&= \frac{1}{N}((n-1)\cdot 2^n + 1) = \frac{1}{N}((N+1)\log_2(N+1) - N) \\
&\approx \log_2(N+1) - 1, \text{ für große } N.
\end{aligned}
$$

Im Mittel verursacht binäres Suchen also etwa eine Kosteneinheit weniger als im schlechtesten Fall.

3.2.3 Fibonacci-Suche

Ein dem binären Suchen analoges Suchverfahren ist die *Fibonacci-Suche*. Dieses Verfahren führt keine (ganzzahlige) Division zur Bestimmung der jeweils mittleren Position des Suchbereichs durch, sondern kommt mit Additionen und Subtraktionen aus. Anstatt den Suchbereich wie beim binären Suchen jeweils strikt in der Mitte zu teilen, nimmt man bei der Fibonacci-Suche eine Teilung entsprechend der Folge der Fibonacci-Zahlen vor, die wie folgt definiert sind:

$$F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2} \quad \text{für } (n \geq 2).$$

Nehmen wir nun der Einfachheit halber an, daß das zu durchsuchende Feld die Länge $F_n - 1$ hat, es sei also $N = F_n - 1$. Dann teilen wir den zu durchsuchenden Bereich entsprechend dem Paar der vorangehenden Fibonacci-Zahlen auf (vgl. Abbildung 3.3).

Die Fibonacci-Suche kann also folgendermaßen beschrieben werden: Vergleiche den Schlüssel des Elements an der Position $i = F_{n-2}$ mit dem gesuchten Schlüssel k. Falls $a[i].key > k$ ist, durchsuchen wir den linken (unteren) Bereich mit $F_{n-2} - 1$ Elementen auf dieselbe Weise. Falls $a[i].key < k$ ist, durchsuchen wir den rechten (oberen) Bereich

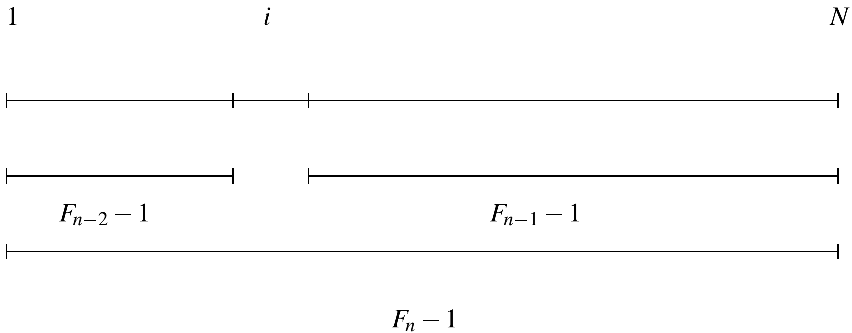

Abbildung 3.3

mit $F_{n-1}-1$ Elementen auf dieselbe Weise. Ist $a[i].key = k$, so endet die Suche erfolgreich. Die Suche endet erfolglos, wenn der im Anschluß an einen Schlüsselvergleich zu durchsuchende Rest des Feldes leer geworden ist.

Um für einen Suchbereich mit Länge $F_j - 1$ die Position des nächsten zu betrachtenden Elements leicht finden zu können, merken wir uns zu jedem Suchbereich jeweils ein Paar von Fibonacci-Zahlen. Zu einem Bereich mit Länge $F_j - 1$ merken wir uns das Paar $(f_1, f_2) = (F_{j-3}, F_{j-2})$.

Nehmen wir also an, der zu durchsuchende Bereich habe die Länge $F_j - 1$ und wir kennen $(f_1, f_2) = (F_{j-3}, F_{j-2})$. Dann wird der Suchschlüssel k als nächstes mit dem Schlüssel des Elements an Position $i = f_2$ verglichen.

Falls $k > a[i].key$ ist, müssen wir rechts weitersuchen. Der zu durchsuchende Bereich hat dann die Länge $F_{j-1} - 1$. Er ist leer, falls $F_{j-1} = 1$ ist, was wir am Wert von f_1 leicht ablesen können. Sobald $f_1 = F_{j-3} = 0$ (und $F_{j-2} = 1$) geworden ist, ist $F_{j-1} = F_{j-3} + F_{j-2} = 1$. Als neues Paar von Fibonacci-Zahlen müssen wir uns bei nichtleerem Bereich das Paar

$$(f_1', f_2') = (F_{j-4}, F_{j-3})$$

merken, das man aus dem alten Paar (f_1, f_2) leicht wie folgt erhält:

$$(f_1', f_2') = (f_2 - f_1, f_1)$$

Falls $k < a[i].key$ ist, müssen wir links weitersuchen. Der zu durchsuchende Bereich hat dann die Länge $F_{j-2} - 1$. Er ist leer, falls $F_{j-2} = 1$, also $f_2 = 1$ geworden ist. Als neues Paar von Fibonacci-Zahlen müssen wir uns bei nichtleerem Bereich das Paar

$$(f_1', f_2') = (F_{j-5}, F_{j-4})$$

merken, das ebenfalls aus dem alten Paar (f_1, f_2) leicht berechnet werden kann.

In der folgenden programmtechnischen Realisierung des Verfahrens Fibonacci-Suche gehen wir davon aus, daß die Fibonacci-Zahlen F_n, F_{n-2}, F_{n-3} explizit gegeben sind, also etwa als Konstanten im Rahmenprogramm der Suchprozedur vereinbart wurden.

```
procedure fibsearch (k : integer);
var i, f1, f2, aux : integer;
    gefunden, nichtgefunden : boolean;
begin
  gefunden := false;  nichtgefunden := false;
  f1 := F_{n-3};  f2 := F_{n-2};  i := f2;
  repeat
    if k > a[i].key  {oberen Bereich durchsuchen}
      then
        if f1 = 0  {Suche beendet}
          then nichtgefunden := true
          else
            begin
              i := i + f1;
              aux := f1;
              f1 := f2 − f1;
              f2 := aux
            end
        else
          if k < a[i].key  {unteren Bereich durchsuchen}
            then
              if f2 = 1 {Suche beendet}
                then nichtgefunden := true
                else
                  begin
                    i := i − f1;
                    f2 := f2 − f1;
                    f1 := f1 − f2
                  end
            else  {k = a[i].key}
            gefunden := true
  until gefunden or nichtgefunden;
  {Ausgabe von i, falls gefunden, sonst Fehlermeldung}
end
```

Wieviele Schlüsselvergleiche werden bei der Suche nach einem Schlüssel k maximal ausgeführt? Ausgehend von einem Suchbereich mit Länge $F_j - 1$ ist die Länge des nächsten zu durchsuchenden Bereichs höchstens $F_{j-1} - 1$. Daher sind zum Durchsuchen eines Bereichs mit Anfangslänge $F_n - 1$ mit Hilfe von Fibonacci-Suche schlimmstenfalls n Schlüsselvergleiche erforderlich. Nun ist

$$
\begin{aligned}
F_n &= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) = \left[\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n\right] \\
&\approx c \cdot 1.618^n, \text{ mit einer Konstanten } c.
\end{aligned}
$$

Für $N+1 = c \cdot 1.618^n$ Elemente benötigt man also $O(n)$ Schlüsselvergleiche im schlechtesten Fall; d.h. die maximal erforderliche Anzahl von Schlüsselvergleichen ist

$$C_{max}(N) = O(\log_{1.618}(N+1)) = O(\log_2 N),$$

also von derselben Größenordnung wie beim binären Suchen. Man kann zeigen, daß auch die im Mittel ausgeführte Anzahl von Schlüsselvergleichen von dieser Größenordnung ist, vgl. dazu z.B. [89].

3.2.4 Exponentielle Suche

Binäre Suche und Fibonacci-Suche setzen voraus, daß man die Länge des zu untersuchenden Bereichs vor Beginn der Suche kennt. Es kann aber Fälle geben, in denen der Suchbereich zwar endlich, aber „praktisch“ unbegrenzt groß ist. In einem solchen Fall ist es vernünftig, zu einem gegebenen Suchschlüssel k zunächst eine obere Grenze für den zu durchsuchenden Bereich zu bestimmen, in dem ein Element mit Schlüssel k liegen muß, wenn es überhaupt ein solches Element gibt. Dieser Idee folgt die *exponentielle Suche*.

Um in einer Liste $a[1], \ldots, a[N]$ mit sehr großem N ein Element mit Schlüssel k zu finden, bestimmen wir zunächst in exponentiell wachsenden Schritten einen Bereich, in dem ein solches Element liegen muß, wie folgt:

$i := 1;$
while $k > a[i].key$ **do** $i := i + i;$

Für das auf diese Weise bestimmte i gilt dann

$$a[i/2].key < k \leq a[i].key.$$

(Dabei wird $a[0] = 0$ angenommen.) Es genügt also, diesen Bereich nach einem Element mit Schlüssel k zu durchsuchen. Weil wir vorausgesetzt haben, daß die Elemente aufsteigend sortierte, verschiedene positive ganzzahlige Schlüssel haben, wachsen die Schlüssel mindestens so stark wie die Indizes der Elemente. Daher wird i in der oben angegebenen **while**-Schleife maximal $\log_2 k$ mal, beginnend beim Anfangswert 1, verdoppelt. Das gesuchte i läßt sich also mit $\log_2 k$ Schlüsselvergleichen bestimmen. Ebenso ist klar, daß der Suchbereich

$$a[i/2], a[i/2+1], \ldots, a[i]$$

maximal k Elemente enthalten kann. Durchsucht man diesen Bereich nun mit Hilfe des Verfahrens binäres Suchen oder Fibonacci-Suche, so werden nochmals $O(\log k)$ Schlüsselvergleiche ausgeführt. Exponentielle Suche erlaubt es also, in einer Folge von N Elementen mit aufsteigend sortierten Schlüsseln nach einem Element mit Schlüssel k stets in $O(\log k)$ Schritten erfolgreich oder erfolglos zu suchen. Das ist immer dann ein sinnvolles Verfahren, wenn k sehr klein im Vergleich zu N ist.

3.2.5 Interpolationssuche

Bei binärer Suche und Fibonacci-Suche hängt die Position des jeweils nächsten inspizierten Elements nur von der Länge des Suchbereichs, nicht aber von den Werten der Schlüssel im Suchbereich ab. Aus dem täglichen Leben weiß man, daß das in manchen Fällen nicht sinnvoll ist. Man denke etwa daran, wie wir üblicherweise nach einem Namen in einem dicken Telefonbuch einer großen Stadt suchen. Suchen wir etwa nach dem Namen „Bayer“, werden wir das Buch weit vorne, suchen wir den Namen „Zimmermann“, werden wir es weit hinten aufschlagen. Wir schätzen also intuitiv die Position des Namens (des Suchschlüssels) aus dem Wert. Diese Idee führt zu einem Suchverfahren, das als *Interpolationssuche* bekannt ist. Man kann es am einfachsten als eine Variante des binären Suchens erklären. Beim binären Suchen haben wir als nächstes zu inspizierendes Element das Element mit Index m gewählt, wobei

$$m = l + \frac{1}{2}(r - l)$$

ist und l und r die linke und rechte Grenze des Suchbereichs bezeichnen. Bei der Interpolationssuche ersetzt man nun den Faktor $\frac{1}{2}$ durch eine geeignete Schätzung für die wahrscheinliche (oder erwartete) Position des Suchschlüssels k:

$$m = l + \frac{k - a[l].key}{a[r].key - a[l].key}(r - l)$$

Natürlich muß man m noch zur nächstkleineren oder -größeren Zahl runden. Es ist sofort klar, daß dies nur dann eine gute Schätzung für die Position des Suchschlüssels im Bereich $a[l], \ldots, a[r]$ ist, wenn die Schlüsselwerte in diesem Bereich einigermaßen gleichverteilt sind. Man kann zeigen (vgl. z.B. [199]), daß Interpolationssuche im Mittel $\log_2 \log_2 N + 1$ Schlüsselvergleiche ausführt, wenn die N Schlüssel unabhängig und gleichverteilte Zufallszahlen sind. Man beachte aber, daß dieser Vorteil der geringen Anzahl von Schlüsselvergleichen durch die größere Komplexität der auszuführenden arithmetischen Operationen leicht wieder verloren geht. Außerdem benötigt Interpolationssuche im schlimmsten Fall linear viele Schlüsselvergleiche, im Unterschied zu allen anderen in Abschnitt 3.2 vorgestellten Suchverfahren, die Sortierung ausnutzen.

3.3 Selbstanordnende lineare Listen

Sind die Zugriffshäufigkeiten für die Elemente linearer Listen sehr unterschiedlich, kann es ratsam sein, die Elemente, auf die häufig zugegriffen wird, möglichst weit vorn und die Elemente, auf die selten zugegriffen wird, am Ende der Liste zu plazieren, und die Liste dann stets linear von vorn nach hinten zu durchsuchen. Leider kennt man aber oft die (relativen) Zugriffshäufigkeiten nicht im voraus, so daß man sie auch bei der Organisation von Listen nicht berücksichtigen kann. Man kann aber versuchen, nach jedem Zugriff auf ein Element die Liste so zu verändern, daß eine künftige Suche nach diesem Element schneller geht. Wir diskutieren in diesem Abschnitt die wichtigsten

Strategien zur Selbstanordnung von Listen, die dieses Ziel verfolgen. Die betrachteten Listen sind im allgemeinen nicht nach Schlüsselwerten sortiert und können sequentiell oder verkettet gespeichert vorliegen.

Folgende drei Strategien sind in der Literatur besonders ausführlich untersucht worden:

MF-Regel (Move-to-front): Mache ein Element zum ersten Element der Liste, nachdem auf das Element (als Ergebnis einer erfolgreichen Suche) zugegriffen wurde. Die relative Anordnung der übrigen Elemente bleibt unverändert.

T-Regel (Transpose): Vertausche ein Element mit dem unmittelbar vorangehenden, nachdem auf das Element zugegriffen wurde.

FC-Regel (Frequency Count): Ordne jedem Element einen Häufigkeitszähler zu, der anfangs 0 ist und die Anzahl der Zugriffe auf das Element speichert. Nach jedem Zugriff auf ein Element wird dessen Häufigkeitszähler um 1 erhöht. Ferner wird die Liste nach jedem Zugriff neu geordnet und zwar so, daß die Häufigkeitszähler der Elemente in absteigender Reihenfolge sind.

Die Wirkung dieser Regeln wird dann besonders klar, wenn die Zugriffshäufigkeiten der Elemente sehr unterschiedlich sind oder die Suchargumente in der Zugriffsfolge stark gebündelt auftreten. Zur Verdeutlichung betrachten wir folgendes Beispiel.

Gegeben sei die aufsteigend sortierte Liste von sieben Schlüsseln $1,2,3,4,5,6,7$. Die erste Zugriffsfolge greift auf die Elemente in der Liste zehnmal nacheinander in der Reihenfolge $1,\dots,7$ zu. Die zweite Zugriffsfolge greift zunächst zehnmal auf 1, dann zehnmal auf 2, usw. und schließlich zehnmal auf 7 zu.

In beiden Zugriffsfolgen wird auf jedes Element der gegebenen Liste zehnmal zugegriffen. Was sind die Kosten, wenn man auf die Elemente etwa nach der MF-Regel zugreift? Es ist üblich, als Kosten (oder Schrittzahl) für den Zugriff auf ein Element, das sich an Position i in der Liste befindet, i anzusetzen. Dann kann man die Kosten für beide Zugriffsfolgen leicht angeben.

Die ersten sieben Zugriffe der ersten Folge benötigen $\sum_{i=1}^{7} i = \frac{7\cdot 8}{2}$ Schritte. Danach befinden sich die sieben Schlüssel in der Anordnung $7,6,5,\dots,1$. Jeder weitere Zugriff der ersten Folge benötigt jetzt genau sieben Schritte, weil das jeweils nächste gesuchte Element ganz am Listenende steht. Als durchschnittliche Kosten pro Zugriff der ersten Zugriffsfolge erhält man also:

$$\frac{\frac{7\cdot 8}{2} + 7\cdot 9\cdot 7}{10\cdot 7} = 6.7$$

In der zweiten Zugriffsfolge benötigt der $(10\cdot i+1)$-te Zugriff jeweils $(i+1)$ Schritte $(0 \leq i < 7)$. Alle anderen Zugriffe benötigen nur einen Schritt, da sich das Element, auf das nach der MF-Regel zugegriffen wird, bereits am Listenanfang befindet. Es ergeben sich in diesem Fall also als durchschnittliche Kosten:

$$\frac{\sum_{i=1}^{7} i + 9\cdot 7\cdot 1}{10\cdot 7} = 1.3$$

Die relative Zugriffshäufigkeit ist in beiden Fällen für alle Schlüssel gleich. Vorabsortierung und statische Anordnung nach abnehmenden relativen Zugriffshäufigkeiten kann also nichts bringen. Die Liste kann irgendwie, muß aber fest angeordnet werden. Dann sind die durchschnittlichen Zugriffskosten $(10\cdot \sum_{i=1}^{7} i)/70 = 4$.

Das zeigt, daß die MF-Regel zu geringeren durchschnittlichen Kosten führen kann als die „beste“ statische Anordnung. Dies ist insbesondere dann der Fall, wenn die Suchschlüssel in der Zugriffsfolge stark gebündelt auftreten.

Das Vorziehen eines Elements an den Listenanfang nach der MF-Regel ist natürlich eine sehr drastische Veränderung, die erst allmählich korrigiert wird, wenn ein „seltenes“ Element „irrtümlich“ an den Listenanfang gesetzt wurde und auf das Element dann lange nicht mehr zugegriffen wird. Die T-Regel ist in diesem Punkte vorsichtiger und macht entsprechend geringere Fehler; die häufig gesuchten Elemente wandern erst ganz allmählich an den Listenanfang. Man kann aber leicht Zugriffsfolgen angeben, so daß Zugriffe nach der T-Regel praktisch überhaupt nichts nützen: Man betrachte etwa eine Folge von Zugriffen, in der man immer wieder auf die letzten beiden Elemente N, $N-1$, N, $N-1$, ... der Liste $1,\ldots,N$ zugreift. Jeder Zugriff verursacht die maximalen Kosten N.

Die FC-Regel sorgt dafür, daß nach jedem Zugriff die Listenelemente nach abnehmender Zugriffshäufigkeit geordnet sind. Diese Regel hat gegenüber den beiden anderen den schwerwiegenden Nachteil, daß man zusätzlichen Speicherplatz zur Aufnahme der Häufigkeitszähler bereitstellen muß. Falls man die Zugriffshäufigkeiten nicht ohnehin aus anderen Gründen mitführt (etwa, um eine Benutzerstatistik aufzustellen), lohnt die Verwendung der FC-Regel also nicht.

In der Literatur sind neben den genannten noch zahlreiche weitere Permutationsregeln zur Selbstanordnung von Listen vorgeschlagen worden. Eine gute Übersicht gibt [76].

Was ist die optimale Strategie? Offenbar ist diese Frage schon deshalb nicht leicht zu beantworten, weil eine allgemein akzeptierte, präzise Fassung des Optimalitätsbegriffs schwierig ist. Der Optimalitätsbegriff muß ja nicht nur unterschiedliche Zugriffshäufigkeiten, sondern auch Clusterungen von Zugriffsfolgen, die sogenannte *Lokalität*, berücksichtigen können. Daher findet man in der Literatur meistens nur asymptotische Aussagen über das erwartete Verhalten der Strategien zur Selbstanordnung für Zugriffsfolgen, die bestimmten Wahrscheinlichkeitsverteilungen genügen, Lokalität in Zugriffsfolgen bleibt unberücksichtigt. Besonders die MF-Regel ist in dieser Richtung intensiv untersucht worden. Es gibt ferner eine Reihe experimentell ermittelter Meßergebnisse für reale Daten. So berichten Bentley und McGeoch [17]: Die T-Regel ist schlechter als die FC-Regel; die MF-Regel und die FC-Regel sind vergleichbar gut, die MF-Regel ist allerdings in manchen Fällen besser.

Man versucht also, die verschiedenen Strategien zur Selbstanordnung von Listen relativ zueinander zu beurteilen. Ein bemerkenswertes theoretisches Ergebnis in dieser Richtung, das das sehr gute, beobachtete Verhalten der MF-Regel untermauert, gelang Sleator und Tarjan [172]. Zur Formulierung ihrer Aussage führen wir zunächst einige Bezeichnungen ein.

Wir denken uns eine Liste von N Elementen gegeben, auf der wir eine Folge s von m Zugriffsoperationen ausführen wollen. Verfahren A sei eine Strategie zur Selbstanordnung, also etwa die MF- oder T-Regel oder irgendeine andere. Mit $C_A(s)$ bezeichnen wir die gesamte Schrittzahl zur Ausführung aller Zugriffsoperationen der Folge s, beginnend mit der anfangs gegebenen Liste. Dabei nehmen wir an, daß der Zugriff auf ein Listenelement an Position i genau i Schritte benötigt; Vorziehen eines Elements, auf das zugegriffen wurde, an eine näher am Listenanfang befindliche Position kostet nichts. Die dazu erforderlichen Vertauschungen benachbarter Elemente nennen wir *ko-*

stenfreie Vertauschungen. Jede andere Vertauschung benachbarter Elemente heißt eine *zahlungspflichtige Vertauschung*; sie wird mit den Kosten 1 belastet. Wir betrachten also nur solche Algorithmen zur Selbstanordnung, die nach dem Zugriff auf ein Element dieses an eine andere Stelle bewegen und sonst alles fest lassen. Die Vertauschung des Elementes mit einem linken Nachbarn ist frei; jede Vertauschung mit einem rechten Nachbarn kostet eine Einheit.

$C_A(s)$ ist also die Gesamtzahl der Schritte zur Ausführung von s ohne zahlungspflichtige Vertauschungen. $F_A(s)$ bezeichne die Anzahl der kostenfreien und $X_A(s)$ die Anzahl der kostenpflichtigen Vertauschungen bei Ausführung von s mit Verfahren A. Für die MF-, T- und FC-Regel gilt natürlich:

$$X_{MF}(s) = X_T(s) = X_{FC}(s) = 0$$

Greift man auf ein Element an Position i zu, so kann man das Element anschließend maximal mit allen $(i-1)$ vorangehenden Elementen kostenfrei vertauschen. Daher muß für jede Strategie A gelten: $F_A(s) \leq C_A(s) - m$. Nun gilt [172]:

Satz 3.2 *Für jeden Algorithmus A zur Selbstanordnung von Listen und für jede Folge s von m Zugriffsoperationen gilt*

$$C_{MF}(s) \leq 2 \cdot C_A(s) + X_A(s) - F_A(s) - m.$$

Dieser Satz besagt grob, daß die MF-Regel höchstens doppelt so schlecht ist wie *jeder* andere Algorithmus zur Selbstanordnung von Listen. Die MF-Regel ist damit nicht wesentlich schlechter als die beste überhaupt denkbare Strategie. Selbst Vorkenntnisse über die Zugriffsverteilung können nicht viel nützen. Sleator und Tarjan [172] beweisen sogar ein noch etwas stärkeres Resultat, da sie auch Einfüge- und Streichoperationen in der Operationsfolge s zulassen.

Der Beweis des Satzes benutzt eine Technik, die als *Bankkonto-Paradigma* bekannt geworden ist. Es dient dazu, die durchschnittlichen Kosten pro Operation für eine beliebige Folge von Operationen nach oben hin abzuschätzen. Eine solche Abschätzung nennt man eine *amortisierte Worst-case*-Analyse. Würde man jede Einzeloperation einer beliebig gewählten Operationsfolge einfach durch die schlechtestenfalls mögliche Schrittzahl abschätzen, würde man im allgemeinen eine unrealistisch schlechte Abschätzung der für eine Folge von Operationen erforderlichen Schrittzahl erhalten. Denn in vielen Fällen benötigen nur sehr wenige Operationen einer ganzen Folge von Operationen den für eine Einzeloperation möglichen Maximalaufwand. Das Bankkonto-Paradigma ist eine Methode zur Ermittlung und Verteilung der anfallenden Gesamtkosten.

Wir ordnen daher jedem bei der Abarbeitung der Zugriffsfolge auftretenden Bearbeitungszustand einen Kontostand zu. Eine Einheit auf dem Konto repräsentiert gewissermaßen eine Kosteneinheit bei der Abschätzung der Gesamtkosten. Genauer: Seien eine Liste L, eine Folge s von m Zugriffsoperationen und ein Algorithmus A zur Ausführung gegeben. Wir wollen den Aufwand bei der Abarbeitung von s nach der MF-Regel $C_{MF}(s)$ mit dem Aufwand $C_A(s)$ bei Abarbeitung von s mit Hilfe von A vergleichen. Dazu lassen wir A und MF die Operationsfolge s gleichzeitig, parallel abarbeiten. Anfangs starten A und MF mit derselben Liste. Nach Ausführung jeder weiteren Operation sind die von A und MF erzeugten Listen im allgemeinen verschieden; dieses Paar von

Listen charakterisiert den bis dahin erreichten Bearbeitungszustand. Wir werden ihm einen Kontostand ϕ zuordnen.

Nun definieren wir als die *amortisierte Zeit* a_l zur Ausführung der l-ten Operation der Folge s die *wirkliche* Schrittzahl (Zeit) t_l zur Ausführung dieser Operation plus die Differenz $\phi_l - \phi_{l-1}$ der Kontostände. Dabei bedeutet ϕ_l den Kontostand nach Ausführung der l-ten Operation und ϕ_{l-1} den Kontostand vor Ausführung der l-ten Operation, also nach Ausführung der $(l-1)$-ten Operation der Folge s. D.h. es ist

$$a_l = t_l + \phi_l - \phi_{l-1}, \text{ für } 1 \leq l \leq m.$$

ϕ_0 ist der Kontostand zu Beginn, d.h. vor Ausführung der Operationsfolge s. Damit gilt:

$$\begin{aligned} \sum_{l=1}^{m} a_l &= \sum_{l=1}^{m} t_l + \phi_m - \phi_0, \text{ also} \\ \sum_{l=1}^{m} t_l &= \sum_{l=1}^{m} a_l + \phi_0 - \phi_m \end{aligned}$$

Wir können also die gesamte (wirkliche) Schrittzahl zur Ausführung der m Operationen der Folge s nach oben abschätzen, wenn es uns gelingt, die amortisierten Kosten a_l für jedes l nach oben abzuschätzen, und wenn wir ϕ_0 und ϕ_m kennen.

Als ersten Schritt müssen wir also jedem Bearbeitungszustand einen Kontostand zuordnen. Bearbeitungszustände sind durch das Paar von Listen, also die erreichte Permutation von Elementen der Liste, charakterisiert, auf der die nächste Zugriffsoperation nach dem Verfahren A bzw. nach der MF-Regel operiert. Wir ordnen daher ganz allgemein zwei Listen L_1 und L_2, die dieselben Elemente in unterschiedlicher Anordnung enthalten, einen *Kontostand* $bal(L_1, L_2)$ wie folgt zu:

$$bal(L_1, L_2) = \text{Anzahl der Inversionen von Elementen in } L_2 \text{ bzgl. } L_1$$

Dabei heißt ein Paar i, j von Elementen eine Inversion in L_2 bzgl. L_1, wenn i in L_2 vor j und i in L_1 nach j auftritt.

Beispiel: Gegeben seien die zwei Listen

$$\begin{aligned} L_1 &: 4,3,5,1,7,2,6 \\ L_2 &: 3,6,2,5,1,4,7 \end{aligned}$$

Dann gilt in L_2: 3 vor 4, 6 vor 2, 6 vor 5, 6 vor 1, 6 vor 4, 6 vor 7, 2 vor 5, 2 vor 1, 2 vor 4, 2 vor 7, 5 vor 4, 1 vor 4, aber in L_1 der Reihe nach jeweils die umgekehrte Relation; alle anderen Paare stehen in L_2 und L_1 in derselben Anordnung. Es ist also $bal(L_1, L_2) = 12$.

Wenn (i, j) eine Inversion von L_2 bzgl. L_1 ist, so ist (j, i) eine Inversion in L_1 bzgl. L_2. Daher ist $bal(L_1, L_2) = bal(L_2, L_1)$, obwohl die Definition des Kontostandes für ein Paar von Listen asymmetrisch formuliert ist. Der Kontostand $bal(L_1, L_2)$ mißt, wieviele Elemente in L_2 „falsch“ stehen, wenn man die Reihenfolge der Elemente in L_1 als die „richtige“ ansieht. Deshalb kann man die Elemente in L_1 und L_2 auch so umnumerieren und umbenennen, daß in L_1 gerade 1, 2, 3, $\ldots, N$ in dieser Reihenfolge auftreten und die Inversionszahl unverändert bleibt.

Wir erläutern dies für die beiden oben angegebenen Listen mit sieben Elementen. Das erste Element 4 in L_1 kommt an Position 6 in L_2 vor; das zweite Element 3 in L_1 kommt an Position 1 in L_2 vor, usw. Statt die Listen L_1 und L_2 zu betrachten, können wir also auch die folgenden nehmen:

$$\begin{array}{lcl} L_1{}' & : & 1,2,3,4,5,6,7 \\ L_2{}' & : & 2,7,6,3,4,1,5 \end{array}$$

Es ist $bal(L_1,L_2) = bal(L_1{}',L_2{}')$, wie man leicht nachprüft.

Nun wollen wir die amortisierten Kosten a_l der l-ten Zugriffsoperation nach der MF-Regel durch die Zugriffskosten auf dasselbe Element nach der A-Regel abschätzen. Der Bearbeitungszustand vor Ausführung der Zugriffsoperation sei charakterisiert durch das Paar L_A und L_{MF} von Listen. Wir können annehmen, daß L_A die Liste $1,2,\ldots,N$ ist und auf das i-te Element i in L_A zugegriffen wird. Der Kontostand vor Ausführung der Zugriffsoperationen ist $bal(L_A,L_{MF})$. Sei k die Position, an der das Element i in der Liste L_{MF} auftritt. Diese Situation wird in Abbildung 3.4 veranschaulicht.

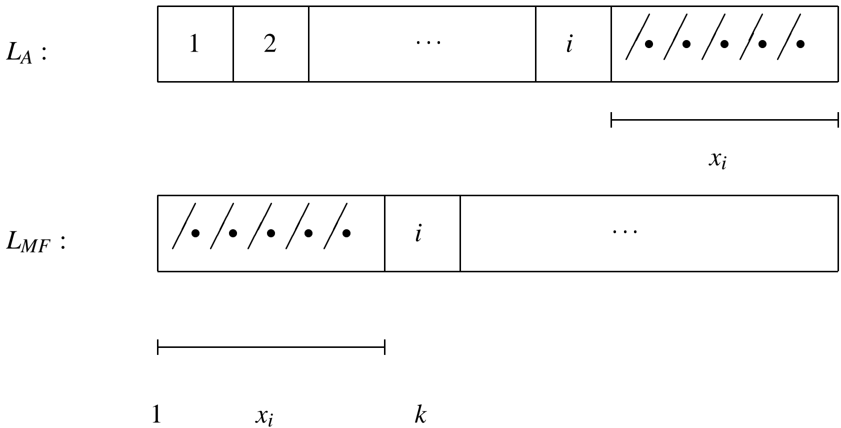

Abbildung 3.4

Sei x_i die Anzahl der Elemente, die i in der Liste L_{MF} vorangehen, aber i in der Liste L_A folgen. (Jedes dieser Elemente ist an einer Inversion in L_{MF} bzgl. L_A beteiligt.) Der Zugriff auf i nach der MF-Regel kostet $t_l = k$ Schritte; durch Vorziehen von i an den Listenanfang entsteht eine neue Liste $L_{MF}{}'$. Die Zahl der Inversionen in $L_{MF}{}'$ bzgl. L_A nimmt offenbar um x_i ab und um genau $k-1-x_i$ Inversionen zu. Ein Zugriff auf i in L_A ohne Vertauschung kostet i Schritte. Jede kostenfreie Vertauschung von i mit einem i in L_A vorangehenden Element verringert die Anzahl der Inversionen in $L_{MF}{}'$ bzgl. der veränderten Liste L_A um 1; mit anderen Worten, jedes Vorziehen von i in L_A um eine Position nach vorn bewirkt, daß es in $L_{MF}{}'$ ein Element j weniger gibt, für das gilt: i geht in $L_{MF}{}'$ j voran, aber i folgt in der veränderten L_A-Liste auf j. Genauso folgt, daß

jede kostenpflichtige Vertauschung in L_A, also jedes Nach-hinten-Schieben von i in L_A um eine Position eine weitere Inversion in L_{MF}' erzeugt. Führt die A-Regel also nach dem Zugriff auf i $F_A(i)$ kostenfreie oder $X_A(i)$ kostenpflichtige Vertauschungen durch, so entsteht eine neue Liste L_A', und es gilt für den neuen Kontostand

$$bal(L_A', L_{MF}') = bal(L_A, L_{MF}) - x_i + (k - 1 - x_i) - F_A(i) + X_A(i).$$

Da die wirkliche Zeit t_l, um auf das Element i nach der MF-Regel zuzugreifen und es an den Listenanfang vorzuziehen, nach Annahme gleich k ist, erhält man als amortisierte Kosten a_l des Zugriffs auf i nach der MF-Regel:

$$\begin{aligned} a_l &= t_l + bal(L_A', L_{MF}') - bal(L_A, L_{MF}) \\ &= k - x_i + (k - 1 - x_i) - F_A(i) + X_A(i) \\ &= 2(k - x_i) - 1 - F_A(i) + X_A(i). \end{aligned}$$

Weil x_i die Anzahl der Elemente ist, die i in L_{MF} vorangehen, aber i in L_A folgen, ist $k - 1 - x_i$ die Anzahl der Elemente, die i in L_{MF} und in L_A vorangehen. Das können aber höchstens $i - 1$ sein. Daher ist $k - x_i \leq i$, und es folgt:

$$a_l \leq 2i - 1 - F_A(i) + X_A(i).$$

Da i die Zugriffskosten (ohne Vertauschungen) nach dem Verfahren A sind, folgt:

$$\sum_{l=1}^{m} a_l \leq 2C_A(s) - m - F_A(s) + X_A(s).$$

Weil das Bankkonto anfangs Null ist, $bal(L, L) = 0$, und das Bankkonto für die nach Ausführung aller m Operationen der Folge s entstehenden Listen L', L'' nicht negativ sein kann, folgt aus der letzten Abschätzung sofort die Behauptung des Satzes:

$$\begin{aligned} C_{MF}(s) &\leq \sum_{l=1}^{m} a_l + bal(L, L) - bal(L', L'') \\ &\leq 2C_A(s) + X_A(s) - F_A(s) - m \end{aligned}$$

□

Sleator und Tarjan zeigen, daß der Beweis dieses Satzes auf jede Heuristik zur Selbstanordnung von linearen Listen ausgedehnt werden kann, die verlangt, daß ein Element an Position k, auf das zugegriffen wurde, nach dem Zugriff um einen festen Bruchteil k/d an den Listenanfang gezogen wird.

3.4 Aufgaben

Aufgabe 3.1

Gegeben sei die Liste $L = 1, 2, 3, 4, 5, 6, 7$ und die Zugriffsfolge s mit 21 Zugriffen:

$$7, 2, 7, 3, 3, 7, 4, 4, 4, 7, 5, 5, 5, 5, 7, 6, 6, 6, 6, 6, 7$$

Vergleichen Sie das Verhalten der MF-und der T-Regel für diese Zugriffsfolge s, indem Sie das folgende Schema ergänzen:

nächstes Element von s	L_{MF}	Zugriffs-kosten nach MF-Regel	L_T	Zugriffs-kosten nach T-Regel	Konto-stand $bal(L_{MF}, L_T)$
—	1,2,3,4,5,6,7	—	1,2,3,4,5,6,7	—	0
7	7,1,2,3,4,5,6	7	1,2,3,4,5,7,6	7	5
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
	Gesamtkosten:	....	Gesamtkosten:	....	

Aufgabe 3.2

Zeigen Sie, daß der im Abschnitt 3.3 bewiesene Satz richtig bleibt, wenn die Operationsfolge s nicht nur Zugriffsoperationen, sondern auch Einfügungen und Streichungen von Elementen in Listen enthält. Um ein Element in eine Liste einzufügen, durchsucht man die ganze Liste vom Anfang bis zum Ende und fügt das Element als neues letztes Element in die Liste ein, wenn es in der Liste nicht schon vorkommt. Die Kosten, ein Element in eine Liste mit Länge i einzufügen, betragen also $i+1$. Entfernen eines Elementes an Position i kostet i Schritte. Unmittelbar nach einer Einfüge- oder Zugriffsoperation können kostenfreie oder kostenpflichtige Vertauschungen vorgenommen werden.

Aufgabe 3.3

Gegeben sei das Feld a mit der folgenden Schlüssel-Belegung:

	1	2	3	4	5	6	7	8
a:	1	2	4	8	16	32	64	128

Man beschreibe die Suche nach dem Schlüssel 34 im obigen Feld a durch Angabe der Folge der ausgeführten Schlüsselvergleiche, wenn als Suchstrategie exponentielle Suche zur Eingrenzung des Suchbereichs mit anschließender linearer Suche angewandt wird.

Aufgabe 3.4

Gegeben sei eine sortierte Liste von 20 Elementen, die in einem Array mit Länge 20 sequentiell abgespeichert sei. Man gebe für jeden beliebigen Suchschlüssel k an, in welcher Reihenfolge die Schlüssel der Listenelemente mit k verglichen werden, wenn die Fibonacci-Suche als Suchverfahren verwendet wird. Dazu stelle man den der Fibonacci-Suche entsprechenden Suchbaum für eine Liste mit Länge 20 dar. Schließlich berechne man explizit die im Mittel beim Durchsuchen der Liste mit 20 Elementen mittels Fibonacci-Suche erforderliche Anzahl von Schlüsselvergleichen, wobei vorausgesetzt wird, daß die relative Zugriffshäufigkeit für alle Elemente gleich groß ist.

Aufgabe 3.5

Geben Sie für ein Paar V_1, V_2 von Suchverfahren aus Abschnitt 3.2 (sequentielle Suche, binäre Suche, Fibonacci-Suche, exponentielle Suche, Interpolationssuche) einen Suchschlüssel k und zwei Zahlenfolgen A_1 und A_2 an, so daß im schlimmsten Fall in A_1 die Suche nach k mit V_1 größenordnungsmäßig schneller ist als mit V_2 (in A_2 mit V_2 schneller ist als mit V_1), falls dies überhaupt möglich ist.