Zeitschrift: Elemente der Mathematik
Herausgeber: Schweizerische Mathematische Gesellschaft
Band: 40 (1985)
Heft: 2

Artikel: Elementare Prinzipien der Informatik
Autor: Gutknecht, J.
DOI: https://doi.org/10.5169/seals-38828

Elementare Prinzipien der Informatik

Wir haben erfahren, dass das Ordnen von Listen ein geeignetes Mittel zur Beschleunigung und Vereinfachung von Suchprozessen ist. Deshalb studieren wir nun Ordnungs- oder *Sortieralgorithmen*. Sie gehören zu den interessantesten und subtilsten Algorithmen überhaupt.
Wiederum beginnen wir mit einer einfachen Aufgabe. Es soll ein Programm erstellt werden, das zwei Elemente a und b ordnet. Falls $a > b$ ist, so müssen die Variablen vertauscht werden. Ein erster Versuch zur Vertauschung von a und b könnte sein

$$b := a; \quad a := b.$$

Dieser scheitert natürlich. Die Schlussbedingung ist nämlich $a = b$. Vertauschen erfordert eine Hilfsvariable u:

$$u := a; \quad a := b; \quad b := u.$$

Wenn wir diese Anweisungsfolge mit *swap(a,b)* abkürzen, stellt sich unser Programm wie folgt dar:

DO $a > b \rightarrow$ *swap(a,b)* OD.

Die Repetition stoppt offensichtlich nach 0 oder 1 Schritt. Wir erhöhen nun die Anzahl der zu ordnenden Variablen, sagen wir auf sechs: a,b,c,d,e,f. Wir wollen sie *an Ort* ordnen, d.h. ohne Verwendung weiterer Variablen (natürlich mit Ausnahme von u). Die Repetition darf sicherlich nicht aufhören, solange zwei benachbarte Variablen in der falschen Beziehung zueinander stehen:

DO $a > b \rightarrow$ *swap(a,b)*
| $b > c \rightarrow$ *swap(b,c)*
| $c > b \rightarrow$ *swap(c,d)*
| $d > e \rightarrow$ *swap(d,e)*
| $e > f \rightarrow$ *swap(e,f)*
OD

Überraschenderweise ist dies bereits das korrekte Programm. Die Abbruchbedingung, d.h. die Konjunktion der Negationen aller Wachen, ist nämlich $a \leq b \leq c \leq d \leq e \leq f$. Wir erinnern daran, dass gemäss Definition der Repetitionsanweisung die Reihenfolge der Vertauschungen durch dieses Programm nicht festgelegt ist!

Die folgende Tabelle zeigt vier Szenarien, nach denen der Prozess bei gegebenem Anfangszustand ablaufen könnte:

a	b	c	d	e	f	a	b	c	d	e	f	a	b	c	d	e	f	a	b	c	d	e	f
6	5	2	1	4	3	6	5	2	1	4	3	6	5	2	1	4	3	6	5	2	1	4	3
5	6	2	1	4	3	6	2	5	1	4	3	6	5	1	2	4	3	6	5	2	1	3	4
5	2	6	1	4	3	6	2	1	5	4	3	6	5	1	2	3	4	6	5	1	2	3	4
5	2	1	6	4	3	6	2	1	4	5	3	5	6	1	2	3	4	6	1	5	2	3	4
2	5	1	6	4	3	6	2	1	4	3	5	5	1	6	2	3	4	6	1	2	5	3	4
2	5	1	6	3	4	6	1	2	4	3	5	1	5	6	2	3	4	6	1	2	3	5	4
2	5	1	3	6	4	1	6	2	4	3	5	1	5	2	6	3	4	1	6	2	3	5	4
2	1	5	3	6	4	1	6	2	3	4	5	1	5	2	3	6	4	1	2	6	3	5	4
2	1	3	5	6	4	1	2	6	3	4	5	1	5	2	3	4	6	1	2	3	6	5	4
1	2	3	5	6	4	1	2	3	6	4	5	1	2	5	3	4	6	1	2	3	5	6	4
1	2	3	5	4	6	1	2	3	4	6	5	1	2	3	5	4	6	1	2	3	5	4	6
1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6

Kommt der Prozess stets nach einer endlichen Anzahl von Repetitionsschritten zu einem Ende? Es genügt, eine Funktion mit ganzzahligen Werten anzugeben, welche von oben beschränkt ist und deren Wert bei jedem Repetitionsschritt echt grösser wird. Die gewöhnliche Summe aller Elemente ist konstant. Wir betrachten daher die *gewichtete Summe* $s := 0a + 1b + 2c + 3d + 4e + 5f$. Die obere Schranke ist s, ausgewertet für die geordnete Liste.

Die Anzahl der Repetitionsschritte hängt offensichtlich vom Grad der anfänglichen Ordnung ab. Der Leser mag herausfinden, ob es reiner Zufall ist, dass in unserem Beispiel alle vier Szenarien die gleiche Zahl von Schritten aufweisen.

Analog zur Situation beim Suchen wurden auch für das Ordnen Algorithmen entwikkelt, welche eine Grössenordnung effizienter sind als die direkten Lösungen. Einen der erfolgreichsten, nämlich Quicksort [6], wollen wir kurz vorstellen.

Am besten können wir die Arbeitsweise von Quicksort erklären, wenn wir annehmen, dass unser Prozessor die Aufgabe des Ordnens der Liste a[0], a[1], ..., a[n − 1] an zwei Mitarbeiter delegieren möchte. Dazu muss der Prozessor die Liste vorbereiten, d. h. sie in zwei unabhängige Teile aufteilen. Unabhängig heisst, dass jeder der beiden Mitarbeiter in der Lage ist, seinen Teil der Arbeit zu verrichten, ohne den andern mit einzubeziehen. Dies ist tatsächlich der Fall, wenn der Prozessor die Liste vorgängig in zwei Teillisten a[0], ..., a[i − 1] und a[i], ..., a[n − 1] aufteilt, so dass jedes Element der ersten nicht grösser ist als jedes Element der zweiten Teilliste.

Dieser Aufteilungsschritt ist nicht sehr aufwendig. Am besten werden unpassende Elemente paarweise ausgetauscht, wie es die untenstehende Figur andeutet. Natürlich ist es wünschenswert, dass beide Teillisten ungefähr dieselbe Grösse aufweisen. Die Hauptschwierigkeit besteht in der Bestimmung eines Bezugselementes x, nach welchem die Einteilung vorgenommen werden kann. Meistens nimmt man, eher aus Verlegenheit, ein Element in der Mitte der Liste.

				x					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
22	56	31	75	49	19	87	38	17	54
	↑							↑	
22	17	31	75	49	19	87	38	56	54
			↑				↑		
22	17	31	38	49	19	87	75	56	54
				↑	↑				
22	17	31	38	19	49 |	87	75	56	54
a[0]				a[i − 1] |		a[i]		a[n − 1]	

Wie im wirklichen Leben hat jeder der beiden Mitarbeiter unseres Prozessors wiederum zwei Assistenten, an welche er die Arbeit delegieren kann. So kehrt also die ursprüngliche Situation wieder. Falls die Teilliste nur aus einem Element besteht, so ist nichts mehr zu delegieren, da die Arbeit schon getan ist. Es ist charakteristisch für solche *rekursiven Verfahren,* dass die Aufgabe eigentlich nie explizit gelöst wird. Vielmehr wird sie sooft delegiert, bis sie verschwunden ist.

Das Ordnen wird also von mehreren autonomen *Prozessen* bewerkstelligt. In der Tat entspricht jedes Delegieren der Aktivierung eines neuen und unabhängigen Prozesses. Im Prinzip könnten alle diese Prozesse gleichzeitig ablaufen. Dies würde offensichtlich einen beträchtlichen Zeitgewinn für das Ordnen der ganzen Liste mit sich bringen. Diese Eigenschaft des Algorithmus kann jedoch nur dann ausgenützt werden, wenn der zugrundeliegende Computer mit einer entsprechenden Zahl von Hardware-Prozessoren ausgestattet ist. Betrachtet man den Fortschritt der Elektronik, im speziellen der *VLSI*-Technologie (Very Large Scale Integration (von elektronischen Bauelementen)), so scheinen solche Computer für die nähere Zukunft nicht unrealistisch.

Der Quicksort Algorithmus ist jedoch auch mit nur einem Prozessor attraktiv. Falls wir Glück haben mit unseren Bezugselementen, so wird bei jeder Delegation der Aufgabe die Grösse der zu ordnenden Liste halbiert. Andererseits stellt sich heraus, dass das Ordnen zweier Listen von halbem Umfang mit weniger Aufwand verbunden ist als das direkte Ordnen der ganzen Liste (sogar wenn man die Vorbereitung mit einbezieht).

Eine zweite Serie von Beispielen

Während wir im vorigen Abschnitt den algorithmischen Aspekt von Computerprogrammen betont haben, stehen nun die *Datenstrukturen* im Vordergrund. Normalerweise ist eine *Datenbank,* welche von einem Computerprogramm verwaltet wird, kein statisches Objekt. Neue Elemente müssen eingefügt und bestehende gelöscht werden können.

Nehmen wir an, dass wir vor die Aufgabe gestellt werden, ein Programm zur Verwaltung einer *Rangliste* eines Abfahrtsrennens zu entwickeln. Jeder Teilnehmer sei durch einen *Datensatz* charakterisiert. Dieser Datensatz enthalte als Komponenten den *Namen* und die im Rennen erreichte *Zeit.*

Wir wollen eine Maximalzahl von n Teilnehmern zulassen. Dazu deklarieren wir als Datenbasis einen Bereich p[1], p[2], ..., p[n] von Datensätzen. Den Namen und die Zeit

des Teilnehmers i bezeichnen wir mit p[i].name und p[i].time. Unser Programm soll die beiden Befehle «Füge neuen Teilnehmer ein» und «Stelle die aktuelle Rangliste dar» ausführen können. Indem wir uns selbst in die Lage versetzen, die Teilnehmer einfügen zu müssen, können wir die prinzipielle Schwierigkeit ausmachen. Entweder gliedern wir die Teilnehmer in der Folge ihrer Ankunft oder in der Folge der erreichten Zeiten in unseren Bereich ein.

Im ersteren Falle füllen wir einfach den Bereich sukzessive auf. Dies bedingt jedoch, dass, jedesmal wenn der Befehl zur Darstellung der Rangliste gegeben wird, der Bereich nach den Zeiten geordnet werden muss. Eine wenig befriedigende Lösung! Im letzteren Fall dagegen müssen wir den Bereich jeweils umordnen, um einen neuen Fahrer an die richtige Stelle bringen zu können. Umordnen bedeutet Verschieben vorhandener Teilnehmer nach hinten, so dass eine Lücke für den neuen Fahrer geschaffen wird. Im schlimmsten Fall (wenn der neue Fahrer die beste Zeit erreicht hat), muss jedes Element verschoben werden.

Dies ist offensichtlich ebenfalls keine effiziente Lösung (auch wenn sich der Schaden in unserem Beispiel in Grenzen hält). Als Ausweg bietet sich das Konzept der *Verkettung* an. Dazu fügen wir unseren Datensätzen p[i] ein neues Feld *next* hinzu, das den Index des (zeitlichen) Nachfolgers enthält. Wir interpretieren next als *Zeiger* zum nächsten Element in der Kette. Ferner kommen wir überein, den Zeiger 0 als das Ende der Kette zu deuten. Schliesslich benötigen wir eine Variable *first,* welche auf das erste Glied zeigt.

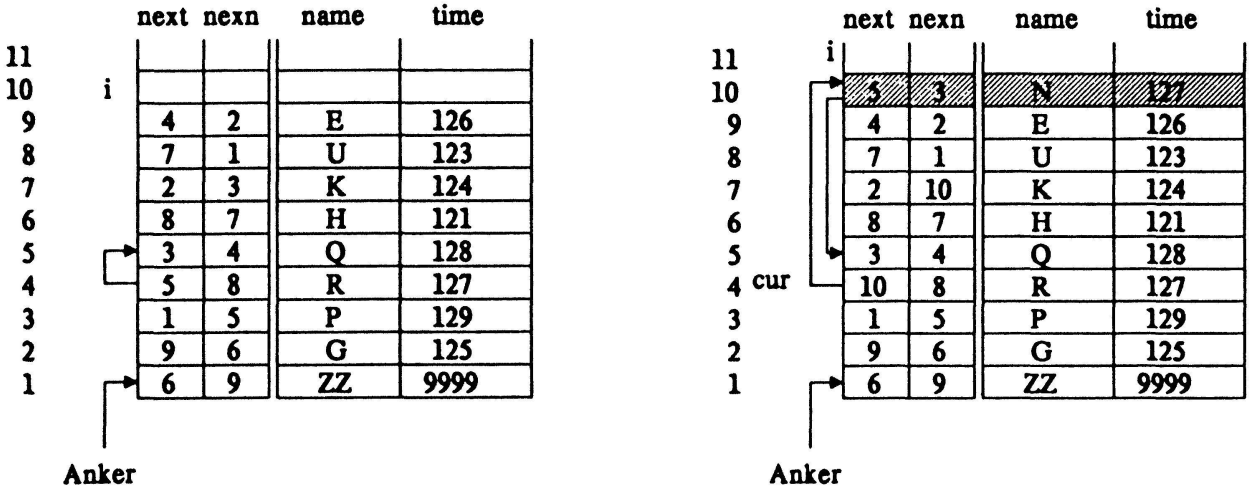
Real werden die Teilnehmer nun hintereinander in den Bereich eingefügt. Der Index i gebe jeweils die nächste freie Position an. Zusätzlich muss jedoch jedes Element richtig in die Kette eingegliedert werden. Nehmen wir an, die Daten des neuen Teilnehmers seien in den Variablen *newname* und *newtime* festgehalten. Dann muss unser Programm die Kette solange durchlaufen, bis ein Glied x erreicht ist, dessen Zeit grösser als *newtime* ist. Wir lassen den Suchprozess ein Glied vorausblicken und nennen das jeweils untersuchte Glied *cur*. Als Schlussbedingung des Durchlaufprozesses ergibt sich somit x = p[cur].next. Das neue Element p[i] wird dann unmittelbarer Vorgänger von x. Durch Anpassung der *next*-Komponenten an der Aufbruchstelle wird die richtige Verkettung wie folgt hergestellt: p[i] zeigt zu x und p[cur] zeigt zu p[i].

Unser Einfügealgorithmus stellt sich somit wie folgt dar:

```
{lies newname and newtime ein}
p[i].name:=newname;  p[i].time:=newtime;
cur:=first;
DO p[p[cur].next].time ≤ newtime → cur:=p[cur].next OD;
p[i].next:=p[cur].next;  p[cur].next:=i;
i:=i + 1
```

Ist das Programm korrekt? Es ist im allgemeinen empfehlenswert, Extremfälle zu betrachten. Hier gibt es zwei. Sie entsprechen den Situationen, in denen das neue Element zum Kopf bzw. zum Abschluss der Kette wird. Der erste Fall wird sicher nicht richtig behandelt, da das Element *first* überhaupt nicht ins Spiel kommt. Das Programm versagt jedoch auch im zweiten Fall. Da die *next*-Komponente des letzten Gliedes 0 ist, wird die Wache undefiniert.

Wir könnten versuchen, den Algorithmus zu korrigieren. Es ist jedoch – wie im allerersten Beispiel des letzten Abschnittes – günstiger, die Datenstruktur anzupassen. Wir führen dazu einen definitiv letzten «Pseudoteilnehmer» ein. Ausserdem, verknüpfen wir diesen Pseudoteilnehmer mit dem momentan ersten Glied in der Kette, schliessen also die Kette zu einem *Ring*. Das Element p[1] stellt eine Art *Verankerung* des Ringes dar. Die Notwendigkeit des Zeigers *first* entfällt somit.
Der Initialisierungsteil unseres Programmes wird so zu



Figur 3. Einfügen eines neuen Elementes in die Zeit- und Namensringkette.

p[1].next:=1; p[1].time:=∞.

Wenden wir uns nun dem zweiten Befehl «Stelle die aktuelle Rangliste dar» zu. Seine Ausführung besteht im wesentlichen im Durchlaufen der Kette:

cur:=p[1].next;
DO p[cur].time # ∞ → {schreibe p[cur].name aus} cur:=p[cur].next OD.

Die Methode des Verkettens hat sich als äusserst geeignet herausgestellt, um den Konflikt zwischen der Reihenfolge des Einfügens der Teilnehmer und der Reihenfolge der erreichten Zeiten aufzulösen. Sie ist praktisch unentbehrlich, falls eine Datenbank gleichzeitig nach verschiedenen Kriterien geordnet werden muss.
In unserem Fall kommt eine zusätzliche Ordnung nach den Namen der Teilnehmer in Frage. Dazu müssen wir lediglich den Datensätzen eine zweite Zeigerkomponente, z. B. *nexn*, hinzufügen. Wir erklären den Pseudoteilnehmer auch lexikographisch zum letzten und schliessen die Namenskette ebenfalls zu einem Ring. Der Einfügealgorithmus muss selbstverständlich um eine entsprechende Anweisungsfolge zur Eingliederung eines neuen Teilnehmers in die Namenskette erweitert werden. Dazu kann der zeitliche Eingliederungsalgorithmus übernommen werden, wobei *next* durch *nexn* ersetzt werden muss. p[1].name muss mit einem «unendlich grossen Namen» initialisiert werden.
Die Darstellungsprozedur für die Namenskette ist dieselbe wie diejenige für die Zeitkette, wo wieder *next* durch *nexn* ersetzt ist. Es sei dem Leser empfohlen, eine Prozedur zu entwickeln, die einen (disqualifizierten) Fahrer aus der Rangliste (nicht aber aus der Namenskette) nimmt.

Figur 3 zeigt die Datenstruktur, die wir gerade besprochen haben. Die zwei Ringketten variieren laufend ihre Grösse und Ordnungsrelation. Man spricht deshalb von einer *dynamischen* Struktur.
Die nächste Illustration, die unsere Beispielreihe beschliesst, geht noch einen Schritt weiter. Die *Textverarbeitung* ist zu einem wichtigen Anwendungsbereich in der nichtnumerischen Informatik geworden. Eine logisch zusammenhängende Folge von Datenelementen heisst *File*. Ein Text ist also nichts anderes als ein File von Zeichen. Ein *Texteditor* ist ein Programm, das einen Text am Bildschirm zeigen und Befehle zu dessen Bearbeitung ausführen kann.
Beschränken wir uns auf die beiden Befehle «Füge einen Textteil ein» und «Lösche einen Textteil». Die Quelle des einzufügenden Textes kann die Tastatur oder ein bestehendes Textfile sein. Nehmen wir zunächst an, dass diese Befehle *direkt* auf den Text, d.h. auf die entsprechende Sequenz von Zeichen wirken. Dann kehrt unser früheres Dilemma wieder, welches seine Ursache in der Diskrepanz zwischen der logischen Reihenfolge und der Reihenfolge der Eingabe hat. Tatsächlich erfordert das Einfügen oder Löschen einer Folge von Zeichen im allgemeinen die Bewegung eines möglicherweise grossen Teiles des ursprünglichen Textes.
Das Verkettungskonzept erweist sich auch hier als Ausweg. Unsere Datenstruktur widerspiegelt die Interpretation des zu verarbeitenden Textes als eine Folge von *Textstükken*. Ein Textstück ist eine Sequenz von logisch aufeinanderfolgenden Zeichen, die auf demselben File gespeichert sind. Wir ordnen jedem Textstück einen *Beschreibungsblock* zu. Dieser gibt das entsprechende Textfile, die Anfangsposition innerhalb des Files und die Länge (in Anzahl Zeichen) des Textstückes an. Da der bearbeitete Text als Kette von Beschreibungsblöcken dargestellt wird, beziehen wir zusätzlich einen Zeiger zu seinem Nachfolger mit ein.
Die Basis unserer Datenstruktur ist somit ein Bereich d[1], d[2], ..., d[n] von Datensätzen d[i] mit Komponenten d[i].file, d[i].pos, d[i].len and d[i].next. Anfänglich besteht der zu bearbeitende Text aus einem einzigen Textstück. Figur 4 zeigt die Entwicklung der Kette während des Verarbeitungsprozesses. Wir bemerken, dass im Gegensatz zum vorigen Beispiel (abgesehen vom Löschen eines disqualifizierten Fahrers), die Kette nun je nach Aktivität wachsen oder *schrumpfen* kann.
Wir führen deshalb ein *Reservoir* von momentan freien Datenblöcken ein. Am einfachsten ist dieses Reservoir ebenfalls als Kette organisiert. Die Variable *free* möge zum Kopf dieser Kette zeigen, welche folgendermassen zu initialisieren ist:

```
free:=1;  i:=1;
DO i # n → d[i].next:=i + 1 OD;
next[n]: = 0
```

Die Prozeduren get(i) and return(i), um einen Datenblock (mit Index i) aus dem Reservoir zu holen bzw. an das Reservoir zurückzugeben, sind zueinander invers:

```
get(i):    IF free # 0 → i:=free;free:=d[i].next FI
return(i):  d[i].next:=free;  free:=i
```

Als abschliessendes Programmbeispiel arbeiten wir einen Algorithmus aus, der einen Textteil löscht. Dieser Teil sei gekennzeichnet durch seine Anfangsposition A und seine

Länge L. Wir nehmen an, dass $A > 0$ und $A + L <$ totale Länge des Textes. Dies ist in der Praxis garantiert, falls das erste und das letzte Zeichen des Textes als Sentinel (Textanfang und Textende) deklariert werden. Die Variable *first* zeige zum ersten Textstück.

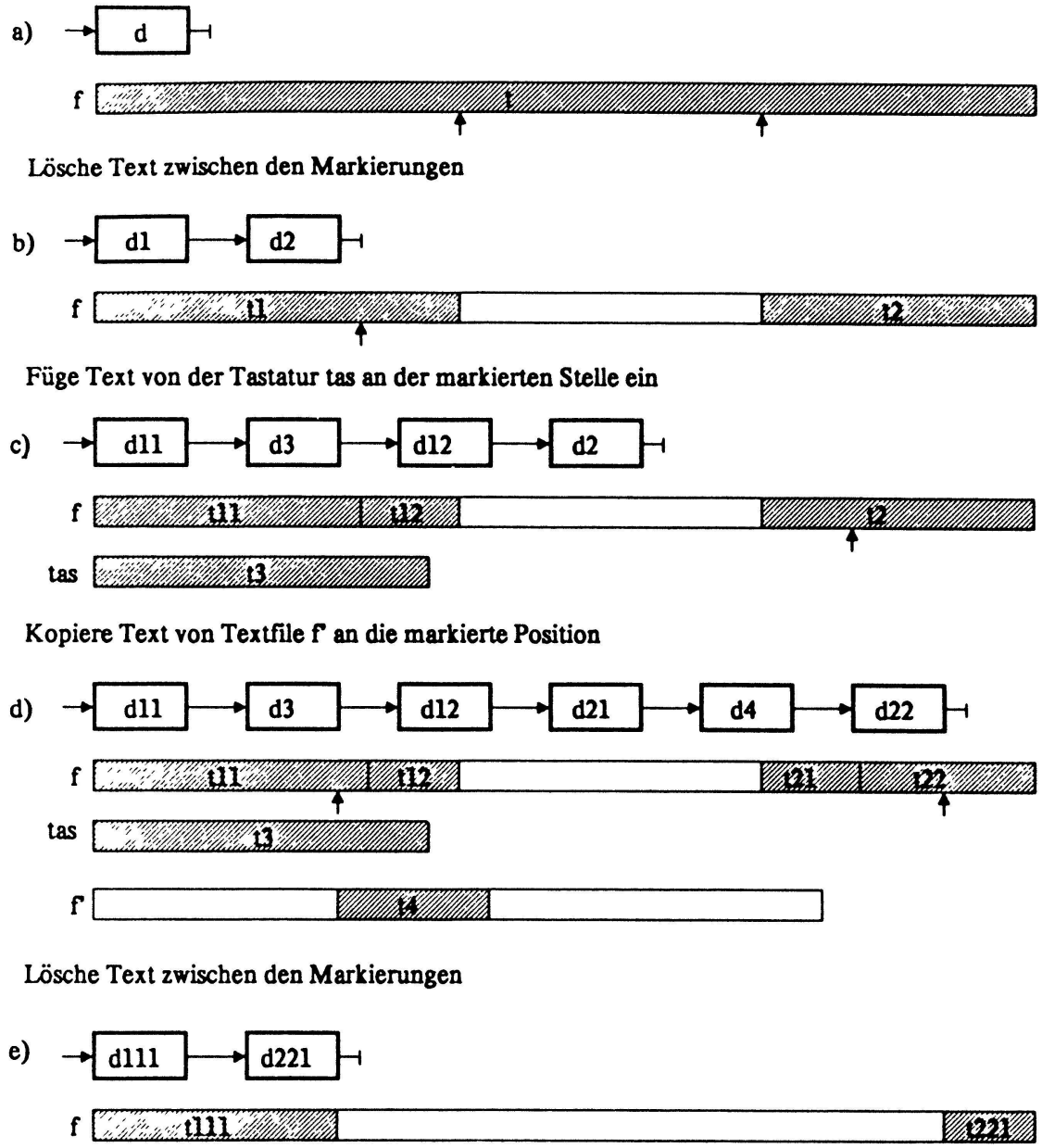
```
B:=A + L;  sum:=0;  a:=first;{sum < A}
DO sum + d[a].len < A → sum:=sum + d[a].len;a:=d[a].next OD;
{sum < A ≤ sum + d[a].len}
get(b);  d[b].next:=d[a].next;  d[a].next:=b;
d[b].len:=sum + d[a].len − A;d[a].len:=A − sum{ > 0};
d[b].file:=d[a].file;  d[b].pos:=d[a].pos + d[a].len;
{sum ≤ B und b = d[a].next}
DO sum + d[b].len ≤ B →
  sum:=sum + d[b].len;d[a].next:=d[b].next;  return(b);  b:=d[a].next
OD;
{sum ≤ B < sum + d[b].len und b = d[a].next}
d[b].len:=sum + d[b].len − B{ > 0};
d[b].pos: = d[b].pos + B − sum;
```

Ein bemerkenswertes Detail dieses Algorithmus ist die Tatsache, dass er keine Textstücke der Länge 0 erzeugt. Das erste Textstück p[a], das in die Löschoperation einbezogen ist, wird (zwischen den beiden Repetitionen) explizit in zwei Stücke p[a] und p[b] aufgespalten. Daher ist der Algorithmus nicht optimal, wenn der zu löschende Teil nicht in einem einzigen Textstück enthalten ist.

Rückblick

Wir haben gesehen, dass Computer Universalgeräte sind, die Daten der verschiedensten Art auf verschiedenste Weise verarbeiten können. Ihre Universalität liegt im Konzept der Software oder Programme begründet. Die Entwicklung korrekter und effizienter Programme ist eine hochgradig mathematische Angelegenheit, selbst dann, wenn die Anwendung nicht-numerischer Art ist. Programmiersprachen und -techniken von grosser Allgemeinheit sind entwickelt worden. Programmiermethoden erleichtern die abstrakte Formulierung von dynamischen Abläufen als statische Texte. Ihre modernen Vertreter unterstützen den Software-Ingenieur im Entwurf von möglicherweise grossen Programmsystemen.

Obwohl die Programmierung eine zentrale Disziplin der Informatik ist, ist sie nicht die alleinige. Die Konzipierung von Datenbanken und von Hardware-Architekturen für Computer-Systeme sind weitere Eckpfeiler. Beide Gebiete haben in letzter Zeit stark an Bedeutung gewonnen: das erste dank der Erschliessung neuer Anwendungsbereiche (z. B. Datenbanken von geometrischen Objekten), das zweite dank dem ungeheuren Fortschritt der elektronischen Technologie (VLSI).



Figur 4. Entwicklung der Text-Beschreibungskette im Laufe der Verarbeitung.

Es ist ein interessantes Faktum, dass Computer in allen diesen Gebieten nicht nur Gegenstand, sondern auch Werkzeug sind. Damit hat sich der Kreis geschlossen.

J. Gutknecht, Institut für Informatik, ETH-Zürich

LITERATURVERZEICHNIS

1 K. Jensen and N. Wirth: Pascal User Manual and Report. Springer Verlag, 1975.
2 N. Wirth: Programmierung in Modula-2. Springer Verlag, 1982.
3 The Ada Programming Language, ANSI/MIL-STD-1815A, Amer. Nat. Std. Inst., 1983.
4 E.W. Dijkstra: A Discipline of Programming. Prentice Hall, 1976.
5 D. Gries: The Science of Programming. Springer Verlag, 1981.
6 C.A.R. Hoare: Quicksort. Comp. Journal 5, No. 1, 1962, 10–15.

 0013-6018/85/060038-08$1.50 + 0.20/0