NEUE ENTWICKLUNGEN IM BEREICH DATENSTRUKTUREN

H. Edelsbrunner *)

Zusammenfassung

In den letzten 15 Jahren hat sich das Programmieren von Computern vom Handwerk zur Kunst bzw. zur Wissenschaft entwickelt. Dabei wurde klar, daß der Algorithmus und die verwendete Datenstruktur das Wesentliche eines Programms ausmachen. Ausgehend von dieser Tatsache begann eine intensive Entwicklung von Datenstrukturen und, damit untrennbar verbunden, von Algorithmen.

Das Ziel dieser Arbeit ist, einen Eindruck dieser Entwicklung von ihren Anfängen bis in die heutige Zeit zu vermitteln. Als Ende der Anfangszeit kann das Aufkommen von sogenannten mehrdimensionalen Datenstrukturen im letzten Jahrzehnt bezeichnet werden. Neben der Darstellung der wichtigsten Teilgebiete der mehrdimensionalen Datenstrukturen wird auch auf neueste Ergebnisse und Blickpunkte eingegangen.

*) Institute für Informationsverarbeitung, Technische Universität Graz, Schießstattgasse 4a, A-8010 Graz, Österreich

1. Einleitung

Bei jeder Erörterung von Datenstrukturen als wissenschaftliche Objekte stellt sich zunächst die Aufgabe, den Begriff zu definieren. Dieses Vorhaben erwies sich in der Vergangenheit als schwieriges Unterfangen und wurde in der einschlägigen Literatur von verschiedenen Autoren verschieden behandelt.

Für stark praxisorientierte Abhandlungen war die Motivation, den Begriff "Datenstruktur" fundiert zu definieren, eher gering. Aus diesem Grund wurde der Begriff "Datenstruktur" sehr informell als die Speicherung von Daten bezeichnet, siehe etwa [Brillinger 72].

Als nützlich erwies sich die Gleichsetzung von Datenstrukturen und gerichteten Graphen, wie es z.B. in [Maurer 74] dargestellt wird. Die Knoten enthalten die eigentlichen Daten und haben ihrerseits ihren Sitz in den Speicherzellen. Die gerichteten Kanten werden entweder explizit durch die Speicherung von Zeigern oder implizit durch Beziehungen der Speicheradressen untereinander realisiert Etwa ist der Graph G in Abbildung 1-1,

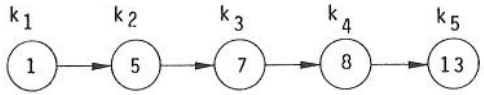

Abbildung 1-1. Graph einer linearen Liste.

bestehend aus fünf Knoten, ein Beispiel der Datenstruktur, die lineare Liste genannt wird. Jeder Knoten enthält einen Wert, hier eine Zahl , und jedem, außer dem letzten Knoten, ist ein Nachfolger mittels eines Zeigers zugeordnet. Die explizite Realisierung von G weist jedem Knoten eine Speicherzelle zu, die den Wert des Knotens, sowie die Adresse der Speicherzelle des Nachfolgeknotens enthält. Ausgehend vom Modell des linearen Speichers, wie es tatsächlich bei

den meisten Computern anzutreffen ist, stellt sich die explizite Realisierung von G etwa wie in Abbildung 1-2 gezeigt, dar.

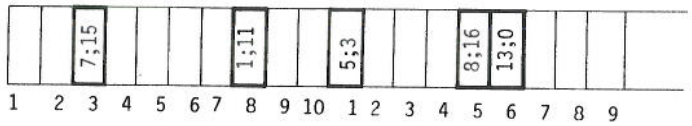

Abbildung 1-2. Explizite Realisierung von G.

Die implizite Realisierung, die wegen ihrer Einfachheit oft vorgezogen wird, sieht für die fünf Knoten fünf hintereinanderliegende Speicherzellen (etwa Zelle 8 bis 12) vor. Die Information, daß z.B. k_2 der Nachfolgeknoten von k_1 ist, bleibt implizit dadurch erhalten, daß k_2 der Speicherzelle nach jener von k_1 zugewiesen wird. Diese Realisierung ist in Abbildung 1-3 dargestellt. Wenn G implizit realisiert wird, nennt man G auch lineares Feld.

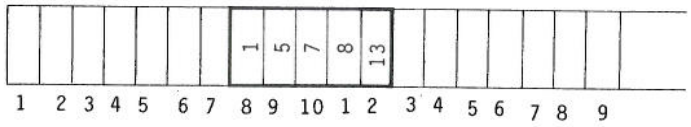

Abbildung 1-3. Implizite Realisierung von G.

In gewissen Situationen ist es sinnvoll, eine Datenstruktur als reinen gerichteten Graphen zu sehen, ohne auf die gespeicherte Information näher einzugehen. Trotzdem bleibt eine Datenstruktur nicht vollständig definiert, wenn nicht auch die Art der Information in den Knoten, sowie die Beziehungen dieser untereinander festgelegt ist. In unserem Beispiel werden ganze Zahlen gespeichert. Sei $w(k_i)$, für $1 \le i \le 5$, die k_1 zugeordnete Zahl, so gilt $w(k_i) < w(k_i+1)$, für $1 \le i \le 4$. Somit sind die Zahlen in G sortiert.

Mit dem Modell gerichteter Graphen für Datenstrukturen kristallisiert sich bereits der Unterschied zwischen dem eher abstrakten Begriff der Datenstruktur und dem eher konkreten Begriff der Realisierung bzw. Speicherstruktur heraus.

In historisch späteren Bemühungen wird versucht, Datenstrukturen noch ein Niveau abstrakter zu beschreiben, siehe etwa [Horowitz 78] und auch [Coleman 78]. Man sieht

gänzlich von der Implementierung ab und definiert die Datenstruktur durch Angabe der Art von zu speichernden Datenelementen und den durchzuführenden Operationen. Wir wollen diesen Begriff streng "abstrakten Datentyp" nennen, um ihn vom Begriff "Datenstruktur", wie er durch Gleichsetzung mit Graphen nahegelegt wird, zu unterscheiden. Die Motivation zu dieser Abstraktion ergibt sich aus dem Wunsch, sich im Prinzip wiederholende Vorgänge oder Strukturen als Primitiva verwenden zu können. Andererseits gibt es für häufig gebrauchte abstrakte Datentypen viele verschiedene konkurrierende Datenstrukturen, die sich oft nur im Detail-Verhalten unterscheiden.

Ein prominentes Beispiel in diesem Zusammenhang ist der abstrakte Datentyp "Wörterbuch". Eine Menge von Werten, etwa Zahlen, soll gespeichert werden und folgende Operationen

"Entscheide ob ein gegebener Wert in der Menge liegt",

"Füge einen weiteren Wert in die Menge ein" und

"Entferne einen Wert aus der Menge"

sollen möglich sein, siehe z.B. [Aho 74].

Alle diese Anstrengungen, den Begriff Datenstruktur zu definieren, wurden geleitet von der intuitiven Vorstellung, daß Computerprogramme aus zwei zusammenspielenden Komponenten bestehen: Dem aktiven Teil, der sich Algorithmus nennt, und der Datenstruktur als passivem Teil (vgl. [Wirth 76]). Der Algorithmus baut sich aus der Datenmenge eine Datenstruktur auf und verwendet diese für weitere Berechnungen. Mitunter modifiziert der Algorithmus auch die Datenstruktur, um sie den sich ändernden Anforderungen anzupassen. Die folgenden Oberlegungen sollen zeigen, daß dieses Modell zwar zum Verständnis der Rollen von Algorithmus und Datenstruktur nützlich sein mag, jedoch nicht eigentlich gültig ist. Die erste Trübung des Modells folgt aus der Beobachtung, daß etwa bei der Obersetzung eines Algorithmus

dessen Instruktionen selbst als Daten für ein anderes Programm, den Kompiler, fungieren. Oberhaupt kann auf tiefster Ebene, in der Algorithmus wie Datenstruktur 01-Sequenzen sind, nicht mehr zwischen beiden unterschieden werden. Für das Modell der Unterscheidung von Algorithmus und Datenstruktur bedrohlich erscheint uns ferner die Tatsache, daß letztere vom Algorithmus ersetzt bzw. simuliert werden kann. Wir wollen dieses Phänomen, das auch in [Pfaltz 77] angesprochen wird, durch ein einfaches Beispiel erläutern:

Es sei M={1,5,7,8,13} im linearen Feld F gespeichert. Der folgende Algorithmus stellt durch sequentielles Durchsuchen von F fest, ob eine Zahl b in M ist oder nicht. Zur Darstellung des Algorithmus bedienen wir uns der Programmiersprache PASCAL, wie in [Jensen 75] beschrieben.

```
var F: array [1..5] of integer;
    b, i, wert: integer;

begin                          {Aufbau der Datenstruktur}
    for i:=1 to 5 do begin read (wert);
                             F [i] := wert
                      end;
                           {Sequentielles Suchen}

    while not EOF do
      begin read (b);
          i := 1;
          while b <> F [i]  and  i<= 5 do  i:= i+1;
          if i= 6 then write (b, " kommt nicht in M vor")
                  else write (b, " kommt in M vor")

    end
end.
```

Tafel 1-4. Programm mit Datenstruktur.

Im ersten Teil des Algorithmus wird das Feld F aufgebaut. Bestehen die Eingebedaten aus den Zahlen 1,5,7,8,13, so sieht das Feld danach wie in Abbildung 1-5 gezeigt aus.

1	5	7	8	13
1	2	3	4	5

Abbildung 1-5. Feld F nach Bespeicherung.

Danach werden Werte b eingelesen und anhand von F wird entschieden, ob b in M ist oder nicht.

Unser Argument ist, daß dieses Zusammenwirken von Algorithmus und Datenstruktur durch einen Algorithmus ersetzt werden kann, der scheinbar ohne Datenstruktur auskommt. Der folgende ist ein solcher Algorithmus:

```
var b : integer ;
begin while not EOF do
    if b=1 or b=5 or b=7 or b=8 or b=13 then
       write (b, " kommt in M vor")
    else write (b, " kommt nicht in M vor")
end.
```

Tafel 1-6. Programm ohne Datenstruktur.

Die auf der Hand liegenden Einwände gegen diesen Algorithmus basieren auf seiner Inflexibilität in bezug auf die Menge, für die er einsetzbar ist: M muß genau fünf Zahlen enthalten und zwar genau die Zahlen 1,5,7,8 und 13. Als Gegenargument führen wir die Existenz von Programmiersprachen an, wie etwa viele Assembler-Sprachen sowie auch LISP, die es ihren Programmen erlauben, sich selbst zu modifizieren.

1.1 Effizienz von Programmen

Zur Lösung eines Problems P mit Hilfe eines Computer-Programms können zumeist verschiedenste Wege eingeschlagen werden. Das heißt, es gibt unterschiedliche Algorithmen und Datenstrukturen, die verwendet werden können. Ein spezielles Programm wird eine Lösung von P genannt, wobei verschiedene Lösungen von P nach mehreren Kriterien miteinander verglichen werden können. Wünschenswerte Eigenschaften von Lösungen sind etwa ihre Korrektheit, ihre Klarheit und ihre Effizienz. Wir wollen nicht näher auf die ersteren beiden Anforderungen an Programme eingehen und voraussetzen, daß alle Programme, mit denen wir uns beschäftigen, ihnen in hohem Maß entsprechen.

Die Effizienz eines Programms wird an der Anzahl von Speicherzellen gemessen, die für die Daten verwendet werden, und an der Zeit, die zur Lösung benötigt wird. Der Bedarf an Speichern und Zeit wird üblicherweise abhängig von der Größe n des Beispiels gemessen, das behandelt wird. Um diese Größen zu veranschaulichen, wenden wir uns dem Programm der Tafel 1-4 zu. Für die Datenstruktur, die aus dem Feld F und den Variablen b, i und wert besteht, werden 8 Speicherplätze benötigt. Die Zeit, die benötigt wird, um F aufzubauen, ist proportional zur Größe von F. Dann spricht man von fünf Schritten, die auszuführen sind. Ebensoviele Schritte genügen, um festzustellen, ob ein Wert b in F ist oder nicht. Man beachte, daß die Begriffe "Speicherplätze" sowie "Schritte" bewußt mit einer gewissen Ungenauigkeit verwendet werden. Die Größe des behandelten Beispiels ist abhängig von der Größe der Menge M, deren Werte in F gespeichert werden. Im betrachteten Fall ist die Größe n des Beispiels gleich 5. Im allgemeinen wird der Aufwand abhängig von der variablen Größe n gemessen. In unserem Fall ist der Speicheraufwand, die Schrittanzahl um F aufzubauen und die Schrittanzahl um für einen Wert b zu entscheiden, ob es in M vorkommt, proportional zu n. Kurz wird dieser Sachverhalt dadurch ausgedrückt, daß der Speicher, wie der

Zeitbedarf in O(n) liegt. In [Knuth 68] kann eine genaue Definition dieser Symbolik gefunden werden.

Die Effizienz von Programmen wird umso wichtiger, je größer die Menge der behandelten Daten ist. So wirkt sich etwa die Tatsache, daß das Programm in Tafel 1-4 sequentiell in einem sortierten Feld sucht,kaum aus, obwohl wesentlich bessere Suchverfahren für diesen Fall existieren. Diese Verfahren zeigen sich dem gezeigten Programm jedoch erst ab einer gewissen Größe des Beispiels überlegen. Allerdings sind in der Praxis kleine Beispiele kaum relevant, da die Vorteile des Computers gegenüber der händischen Verarbeitung primär in der Möglichkeit, große Datenmengen zu behandeln, liegt.

An dieser Stelle kommt die Signifikanz von Algorithmen und Datenstrukturen ins Spiel, die für die Effizienz von Programmen hauptverantwortlich sind. Einerseits sollte ein Algorithmus seine Aufgabe bei gegebener Datenstruktur mit geringem Aufwand bewältigen; andererseits sollte die Datenstruktur diese Tendenz dadurch unterstützen, daß sie die Daten im Hinblick auf die auszuführenden Operationen strukturiert.

2. Anfänge der Datenstrukturen

In diesem Kapitel werden die Anfänge der wissenschaftlichen Untersuchung von Datenstrukturen überblicksmäßig dargestellt. Geschichtlich setzen wird die Jahre 1973 und 1974 als die letzten der Anfangsphase fest. Das Wissen über Datenstrukturen wurde damals in [Knuth 68] , [Knuth 69] und [Knuth 73] recht umfassend zusammengefaßt. Etwas weniger ins Detail gehende Werke, die einen guten Eindruck dieses Wissens vermitteln, sind [Aho 74] und [Maurer 74].

Hauptsächlich konzentrierten sich die Untersuchungen in diesen Anfängen auf Probleme für eindimensionale Daten, d.h.für Daten, die mit jeweils einem Schlüssel (etwa einer Zahl) versehen sind. Für die Datenstruktur ist nur der Schlüssel eines Datums relevant. Vereinfachend kann daher angenommen werden, daß der Schlüssel selbst das Datum ist. Die wichtigsten Probleme für solche Daten waren einerseits das Sortieren von Datenmengen und andererseits das Suchen in solchen Datenmengen. Zur Speicherung wurden lineare Listen und Bäume verwendet. Diese beiden Gruppen von Datenstrukturen umfassen bis heute die wichtigsten Vertreter ihrer Art.

Wir greifen aus dem Material der Anfangszeit der Datenstrukturen jene Teile heraus, die uns am wichtigsten erscheinen. Dazu gehört sicherlich das Problem Mengen von Zahlen, die in linearen Feldern gespeichert sind, zu sortieren.

Sei etwa M eine Folge von n ganzen Zahlen, die unsortiert in einem linearen Feld F gespeichert ist. D.h. F besteht aus n Knoten, wobei jede Zahl in M genau einem Knoten zugeordnet ist. Als Vereinbarung von F in einem PASCAL Programm schreibt man var F:array [1..n] of integer. M sortieren heißt nun die Zahlen so in F umzuordnen, daß die Zahl F [i], die dem i-ten Knoten von F zugeordnet ist, kleiner oder gleich F [j] ist, für i<j. Ein spezielles Verfahren, das F sortiert, heißt"Sortieren durch Verschmelzen". Der Vorgang, zwei sortierte Folgen von Zahlen zu einer

einzigen sortierten Folge zu kombinieren, wird Verschmelzen genannt. F wird nun durch oftmaliges Verschmelzen von sortierten Teilfolgen sortiert. Genauer wird zunächst die linke und dann die rechte Hälfte von F rekursiv sortiert. Zum Abschluß werden die beiden (sortierten) Folgen verschmolzen. Tafel 1-2 zeigt diese Idee in ein PASCAL-Programm umgesetzt.

```
procedure SORTVERSCH (i, j: integer);
var h: integer;
if i<j          then begin
                     h := (i+j)/2;
                     SORTVERSCH (i, h);
                     SORTVERSCH (h+1, j);
                     VERSCHMELZE (i,h,j)
                     end.
```

Tafel 2-1. Sortieren durch Verschmelzen.

Das Sortierverfahren kann jedoch nur funktionieren, wenn auch eine Prozedur VERSCHMELZE zur Verfügung steht, welche zwei sortierte Teilfolgen (nämlich F [i..h] und F [h+1..j]) verschmilzt. Wir verzichten auf die explizite Darstellung dieser Prozedur und begnügen uns mit dem Hinweis, daß z.B. in [Wirth 76] ein solches Verfahren beschrieben wird, dessen Zeitbedarf in O(j-i) liegt. Sei S(n) die Zeit, die benötigt wird, um mit dem angegebenen Verfahren n Zahlen zu sortieren, und sei V(n) der Zeitaufwand, um zwei sortierte Teillisten mit insgesamt n Zahlen zu verschmelzen.
Dann gilt:

$$S(n) = 2\,S(n/2) + V(n) =$$
$$= 2\,S(n/2) + O(n) = O(n\log n).$$

Das vorgestellte Sortierverfahren dient uns als Mittel, um einige wichtige Aspekte von Algorithmen und Datenstrukturen explizit zu erörtern.

Zunächst stellen wir fest, daß das Verfahren n Zahlen in optimaler Zeit sortiert. Das bedeutet, daß die einfache Datenstruktur "lineare Liste" genügt, um optimal zu sortieren. Man beachte in diesem Zusammenhang, daß dasselbe Verfahren in O(nlogn) Zeit sortiert, auch wenn die Liste explizit realisiert ist. Es wird also kein Nutzen aus den zusätzlichen Beziehungen zwischen den Knoten, die durch die implizite Realisierung der linearen Liste ausgenützt werden könnten, gezogen.

Das Verfahren stellt ein klassisches Beispiel der rekursiven Programmierung dar: Das Programm wird von ihm selbst aufgerufen, jedoch mit veränderten Parametern. Die rekursive Darstellung von Algorithmen und Datenstrukturen ermöglicht eine sehr prägnante Ausdrucksweise, welche Strukturen explizit sichtbar macht. Im speziellen sortiert das angegebene Verfahren nach dem Prinzip der Aufspaltung des Gesamtproblems in zwei Hälften, rekursives Sortieren der beiden Hälften und abschließendes Kombinieren der beiden sortierten Listen. Der Anwendungsbereich dieses sogenannten Divide-and-Conquer-Prinzips geht weit über das Sortieren von Zahlenfolgen hinaus.

Wir haben oben erwähnt, daß durch die implizite Realisierung einer linearen Liste zusätzliche Beziehungen unter den Knoten ausgenutzt werden können. Die lineare Liste ordnet jedem Knoten nur seinen Nachfolger zu. In einem zusammenhängenden Feld können durch Indexrechnung aber auch durchaus andere Knoten direkt erreicht werden. So etwa definieren wir als Söhne eines Knotens F [i] die beiden Knoten F [2i] und F [2i+1], falls $2i \leq n$ bzw. $2i+1 \leq n$. F wird eine implizite Realisierung einer Halde genannt, falls F [i] < F [2i] und F [i] < F[2i+1]. Als Datenstruktur stellt sich eine Halde für die Zahlen 1,3,7,8,9,12,17,28,30, 31,32,33,34,35, wie in Abbildung 2-2 gezeigt, dar.

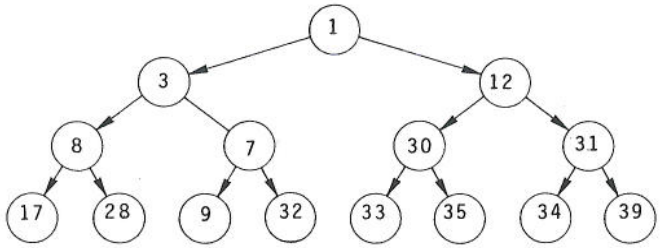

Abbildung 2-2. Halde.

Die Eigenschaften von Halden sind (1), daß die kleinste Zahl in konstanter Zeit (in F [1]) erreichbar ist und (2), daß die Operation "Minimum entfernen" und "Halde wieder herrichten" in O(logn) Zeit mit konstantem zusätzlichem Speicheraufwand ausgeführt werden kann. Dies führt zu einem Sortierverfahren, daß optimal in O(nlogn) Zeit sortiert, jedoch im Unterschied zum Sortieren durch Verschmelzen nur konstanten zusätzlichen Speicher benötigt. Z.B. [Aho 74] beschreibt die Einzelheiten dieses Verfahrens.

Weiters sind lineare Felder ausgezeichnet für das Suchen von Werten b in einer Menge M geeignet. (b suchen heißt feststellen, ob b in M enthalten ist oder nicht). Sind z.B. die Zahlen von M sortiert in einem linearen Feld F gespeichert, kann das sogenannte "binäre Suchen" angewendet werden. Dieses Verfahren beruht essentiell auf der Ausnutzung der impliziten Speicherungen von F, da in konstnter Zeit auf das mittlere Element einer zusammenhängenden Teilfolge von F zugegriffen wird. Tafel 2-3 zeigt das Verfahren als PASCAL-Programm realisiert.


```
var F : array [1..n] of integer;
    i,j,m,b : integer;
begin while not EOF do
        begin read (b);
            i := 1; j := n;
                        {i und j definieren die be-
            trachtete Teilfolge, m ist der Index des
            mittleren Elements.}
            repeat begin m := (i+j)/2;
                    if b<F[m] then j := m-1
                              else i := m+1
                end
            until b = F[m] or i>j;
            if b = F[m] then write (b, " kommt in M vor)
                else write (b, " kommt nicht in M vor)
    end

end.
```

Tafel 2-3. Binäres Suchen.

O(logn) Zeit genügt,um mit diesem Verfahren einen Wert b zu suchen. Außerdem kann gezeigt werden, daß kein schnelleres Verfahren existiert. Zu beachten ist, daß der Algorithmus bei einer explizit realisierten linearen Liste nicht funktioniert, weil wiederholt auf mittlere Elemente von Teilfolgen zugegriffen wird. In gewissem Sinn wird durch das Verfahren eine hierarchische Struktur der Knoten definiert. Der erste Knoten, der für das Verfahren von Bedeutung ist, ist der (n+1)/2-te, also der mittlere der Liste. Wir nennen aus diesem Grund diesen Knoten die Wurzel W. Dann wird entweder der mittlere der Teilfolge links oder rechts von der Wurzel besucht. Wir nennen diese beiden Knoten die Wurzel der linken, bzw. rechten, Teilfolge und auch den linken, bzw. rechten, Sohn von W. Analog werden jedem Knoten bis zwei Söhne zugeordnet und es ergibt sich als Datenstruktur ein sortierter binärer Baum. Abbildung 2-4

zeigt ein sortiertes lineares Feld und den sortierten binären Baum, dessen implizite Darstellung das Feld ist.

1	3	7	8	9	12	17	28	30	31	32	33	34	35	39

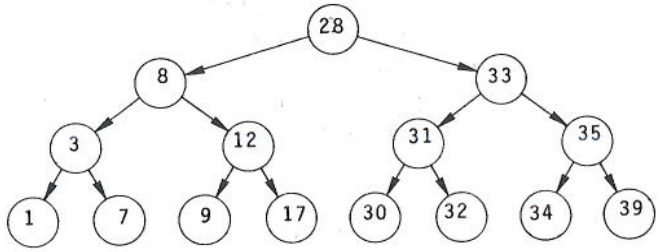

Abbildung 2-4. Lineares Feld und binärer Baum.

Ein Vergleich mit der Halde, dargestellt in Abbildung 2-2 zeigt, daß keine Unterschiede in den Graphen der beiden Datenstrukturen bestehen. Sehr wohl jedoch bestehen Unterschiede in den Beziehungen der gespeicherten Zahlen. Während in der Halde die Zahl eines Knotens kleiner gleich den Zahlen in dessen Söhnen ist, gilt beim sortierten binären Baum, daß die Zahl in einem Knoten größer gleich der Zahl im linken Sohn und kleiner gleich der Zahl im rechten Sohn ist.

In unseren Ausführungen ist der Eindruck entstanden, daß die implizite Realisierung der linearen Liste, der Halde und des sortierten binären Baumes der expliziten Realisierung dieser Datenstrukturen überlegen ist. Diese Situation ändert sich grundlegend, wenn die gespeicherte Menge dynamisch aufrechterhalten wird, d.h., wenn Zahlen in die Menge neu aufgenommen werden, oder Zahlen aus der Menge entfernt werden. Sei etwa die Menge M in einer linearen Liste L gespeichert. Ist L implizit realisiert und soll eine Zahl b neu eingefügt werden, so sind folgende Schritte notwendig:

1. Bestimme kleinstes i sodaß $b \leq L[i]$.
2. Verschiebe die Teilfolge von L[i] ab um eine Position nach rechts.
3. L[i] := b.

Die aufwendigst Aufgabe wird in Schritt 2 verrichtet, der O(n) Zeit benötigt. Ist L explizit realisiert, so sind folgende Operationen zielführend:

1. Bestimme den ersten Knoten p, dessen Zahl größer gleich b ist.
2. Erzeuge einen neuen Knoten q, der b speichert und füge q vor p in die gekettete Liste ein.

Während Schritt 2 in konstanter Zeit durch Adjustierungen von Zeigern ausführt werden kann, benötigt Schritt 1 O(n) Zeit, da keine schnelleren Suchverfahren für explizit realisierte lineare Listen existieren.

Eine bessere Lösung ergibt sich durch Verwendung von sortierten binären Bäumen. Sei B ein sortierter binärer Baum, der M speichert. Ein Folge $p_0, p_1, \ldots p_k$ von Knoten aus B heißt ein Ast (der Länge k), falls p_i ein Sohn von p_{i-1} ist, für $1 \leq i \leq k$. Die Höhe h(B) von B ist die Länge des längsten Astes in B. Durch ein Verfahren analog dem binären Suchen (siehe Tafel 2-3), kann in O(h(B)) Zeit eine Zahl gesucht werden. Ebenso kann in dieser Zeit für eine neue Zahl b ein Knoten p in B bestimmt werden, sodaß b in B eingefügt werden kann, indem ein neuer Knoten q an p angehängt wird. Somit ist die Komplexität der Operation eine Zahl einzufügen proportional zur Höhe von B. Im günstigsten Fall gilt $h(B) = \lceil \log_2 n \rceil$, wie etwa für den Baum in Abbildung 2-4, im schlechtesten Fall jedoch h(B) = n-1, wenn B in eine lineare Liste ausartet.

Die naheliegende Frage, ob die Höhe von B mit geringem Aufwand niedrig gehalten werden kann, ist positiv zu beantworten. So zeigten [Adelson-Velskii 62] bereits 1962, daß durch Umstrukturierungen in O(logn) Zeit pro Einfügung und Ent-

fernung die Höhe von B in O(logn) gehalten werden kann. Für den Algorithmus wird eine spezielle Klasse von binären Bäumen (die sogenannten AVL-Bäume) definiert, deren Höhe für n gespeicherte Werte in O(logn) ist. Der binäre Baum B wird nach jeder Einfügung und Entfernung so umstrukturiert, daß er in der Klasse bleibt. Später sind weitere Algorithmen, die andere Klassen von sortierten Bäumen verwenden, entdeckt worden. Beispiele sind 2-3 Bäume in [Aho 74], gewichtsbalanzierte Bäume in [Nievergelt 73] und Bruderbäume in [Maurer 76]. Als gemeinsames Ergebnis liefern alle diese Baumstrukturen und Strukturier-Algorithmen für das Wörterbuch Probleme:

> Eine Menge M von Zahlen kann in O(logn) Zeit pro Einfügung und Entfernung aufrechterhalten werden, sodaß O(logn) Zeit genügt, um eine gegebene Zahl zu suchen. Mit n ist die laufende Anzahl von Zahlen in M bezeichnet.

Damit haben wir einen Teil der Ergebnisse aus den Anfängen der Datenstrukturen skizziert. Der Vollständigkeit halber besprechen wir noch kurz drei weitere Methoden, die etwas abseits unseres Interesses liegen, jedoch ohne Zweifel wichtig für das Gebiet Datenstrukturen sind:

Sogenannte Hash-Verfahren erlauben uns, eine Menge von n Zahlen so abzuspeichern, daß der Suchaufwand konstant im Erwartungswert ist. Im schlechtesten Fall kann jedoch Zeit proportional zu n nötig sein. Die Zahlen werden gestreut gespeichert, das heißt, mittels einer Funktion H wird jeder Zahl z der Index des Elementes eines linearen Feldes zugeordnet, das z speichert. Schwierigkeiten entstehen, wenn zwei Zahlen dasselbe Element des Feldes zugewiesen wird. Ausführliche Literatur zu Hash-Verfahren im allgemeinen und der Behandlung der angeschnittenen Schwierigkeiten im speziellen kann in [Knuth 73] gefunden werden.

Eine ähnliche Idee führt zu Sortierverfahren durch Fachverteilen, die für viele praktische Probleme den Verfahren, die nur mit Vergleichen arbeiten, vorzuziehen sind. Dabei werden n Zahlen zunächst gestreut gespeichert, wobei die Streufunktion H mit der Ordnungsrelation der Zahlen konform geht. Dann werden die Zahlen durch einmaliges Durchlaufen des zur Speicherung benutzten linearen Feldes wieder eingesammelt. Im allgemeinen ergeben sich nach so einer Phase kleine unsortierte Teilmengen von M, die jedoch untereinander sortiert sind. Durch mehrmaliges Wiederholen einer Phase bei jeweils veränderter Streufunktion H kann M vollständig sortiert werden. Wir verweisen auf [Wettstein 72], der einige Versionen des Sortierens durch Fachverteilen, das er Sortieren durch Adressrechnung nennt, beschreibt.

Besonders bei numerischen Berechnungen treten Daten häufig in Form von Matrizen auf. Sei etwa A eine zweidimensionale Matrix mit m Zeilen und n Spalten. Eine häufige Operation bei solchen Matrizen ist das Zugreifen auf ein Element $A_{i,j}$ durch Angabe seiner Indizes i und j. Die häufigst verwendete Speicherung von A ist die zeilenweise Einbettung in ein lineares Feld F[1.. m*n], d.h. die ersten n Knoten von F enthalten die erste Zeile von A, die zweiten n Knoten die zweite Zeile, etc. Der Knoten von F mit Index (i-1)*m+j enthält die Zahl $A_{i,j}$. Diese Art Matrizen zu speichern kann ohne Schwierigkeiten auf höherdimensionale Matrizen ausgeweitet werden. Bei häufig auftretenden schwach besetzten Matrizen, bei denen also die meisten Werte gleich-0 sind, verwendet man auch gerne die sogenannte verdichtete Speicherung: Nur die Werte ungleich Null werden mit Angabe ihrer Indizes abgespeichert.

Obwohl bei Matrizen von höherdimensionalen Objekten die Rede ist, bedeutet dies nicht, daß hier eine Erweiterung der Speicherung von Zahlen (also eindimensionalen Objekten) auf Tupeln von Zahlen (in gewissem Sinn mehrdimensionalen Objekten) stattgefunden hat. Solche Tupeln kommen in ver-

schiedensten praxisorientierten Gebieten der Informatik vor: z.B. in Datenbanken, die Anfragen zulassen, welche gleichzeitig aufmehrere Attribute der gespeicherten Daten bezug nehmen. Ein Attribut eines Datums, das für eine Anfrage relevant ist, wird ein Schlüssel des Datums genannt.

Ein überstrapaziertes Beispiel in diesem Zusammenhang ist eine Datenbank, die Informationen von Firmenangestellten speichert. Ein Datum besteht z.B. aus

dem Namen,
dem Geburtsdatum,
dem Geschlecht,
dem monatlichen Brutto-Bezug,

und vielleicht aus weiteren Informationen. Ein Beispiel einer Anfrage, die auf mehrere Schlüssel bezug nimmt, ist die Aufgabe, eine Liste jener Angestellten zu erstellen, die durch die Eigenschaften

jünger als 25 Jahre,
weiblich,
S 15.000 Mindestbezug

bezeichnet sind.

Ansätze zur Lösung ähnlicher mehrdimensionaler Probleme sind in [Knuth 73] zu finden. Die beschriebenen Methoden beschränken sich auf offensichtliche Datenstrukturen, wie etwa invertierte Files. Hinter diesem Namen verbirgt sich eine lineare Liste, deren Knoten mehrfach, jedoch jeweils linear, verkettet sind. Z.B. kann die Liste für jeden Schlüssel sortiert und explizit realisiert werden, wie in Abbildung 2-5 für einige Personen in der beschriebenen Datenbank gezeigt wird.

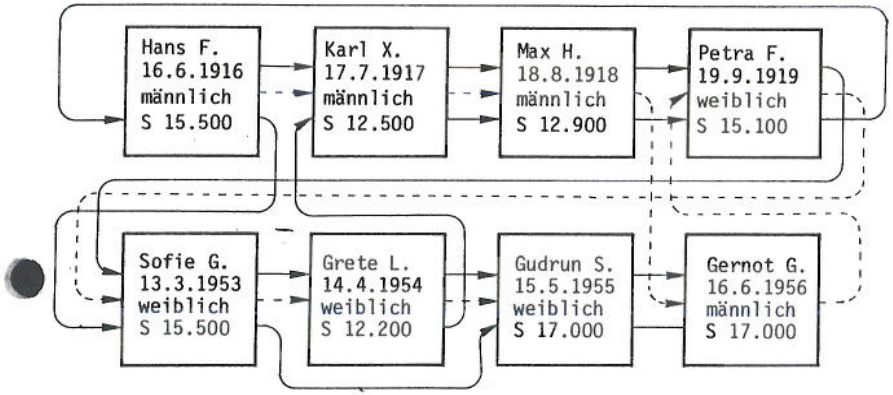

Abbildung 2-5. Invertierter File.

Mehrdimensionale Probleme werden erst seit etwa 10 Jahren eingehender untersucht. Das nächste Kapitel ist eine Darstellung dieser Bemühungen gewidmet.

3. Mehrdimensionale Datenstrukturen

Mehrdimensionale Daten treten nicht nur bei Datenbanken auf, sondern auch in Gebieten wie Computer Graphik, Mustererkennung, Faktorenanalyse, computerunterstütztes Entwerfen, etc. Um eine Behandlung der diversen mehrdimensionalen Probleme unabhängig vom jeweiligen Bereich, aus dem sie stammen, zu ermöglichen, ist zunächst die einheitliche Darstellung wichtig. Diese ist durch die geometrische Interpretation gegeben, die das Verstehen der Probleme erleichtert.

Das in Kapitel 2.0 angeschnittene Datenbank-Problem läßt sich beispielsweise wie folgt interpretieren: Für eine Anfrage sind drei Attribute wichtig, daher enthält jedes Datum drei Schlüssel. Jeder Schlüssel wird unter Aufrechterhaltung der Ordnung, die durch die Menge, aus der er stammt, induziert wird, als reelle Zahl interpretiert. Bei den Schlüsseln "Geburtsdatum" und "Brutto-Bezug" ist eine solche Interpretation offensichtlich. Um den Schlüssel "Geschlecht" in die reelle Zahlengerade einzubetten, werden nur zwei Zahlen, etwa 0 und 1, verwendet.

3.1 Orthogonale Objekte

Jedes Datum wird von einem Punkt p im dreidimensionalen reellen Raum mit Koordinaten p_x,p_y und p_z dargestellt. Eine Anfrage analog dem Datenbank-Beispiel fragt nach allen Punkten $p=(p_x,p_y,p_z)$, sodaß

$$p_x \geq 1958.0101,$$
$$p_y = 1$$
$$p_z \geq 15$$

gilt. Durch die Anfrage wird demnach ein Bereich im Raum spezifiziert, und alle Punkte der gespeicherten Menge in diesem Bereich sind anzugeben. Genauer wird für jede Koordinate ein Intervall spezifiziert, in welchem sie liegen muß. In diesem Fall versteht man unter einem Bereich das kartesische Produkt von drei Intervallen auf den Koordinatenachsen.

In dieser geometrischen Einkleidung wurde das Problem, genannt Bereich Suchproblem, in der Vergangenheit sehr eingehend untersucht. Der Einfachheit halber sei M eine Menge von n Punkten in der Ebene. M soll in einer Datenstruktur gespeichert werden, sodaß für einen Bereich b die Punkte von M in b rasch ausgegeben werden können.

Bereit Knuth [K3] beschrieb eine einfache Lösung des Problems, die auf der Speicherung der Punkte in den Zellen eines orthogonalen Gitters basierte. Sei t die Anzahl der Punkte, die bei einer Anfrage ausgedruckt werden. Im schlechtesten Fall gilt t=n und somit, daß bei jeder Speicherung von M O(n) Zeit für das Beantworten einer Anfrage benötigt wird. Diese grobe Analyse läßt keine bessere Datenstruktur zu als jene, die die Punkte unsortiert in eine lineare Liste speichert.

Um Unterschiede zwischen dieser Lösung und solchen, die bei kleinem t schnell arbeiten, sichbar zu machen, ging man dazu über, den Anteil des Zeitaufwandes abhängig von n vom Anteil abhängig von t zu trennen. So kann z.B. mit Hilfe des sogenannten kd-Baumes für M eine Suchzeit in $O(\sqrt{n}+t)$ bei Speicherbedarf in O(n) erreicht werden, siehe [Bentley 75] und [Lee 77a]. Im folgenden werden der kd-Baum und andere Datenstrukturen beschrieben. Auf die Angabe der Algorithmen und der Vorführung der Analysen wird aus Platzgründen zumeist verzichtet.

Der Einfachheit halber nehmen wir an, daß keine zwei Punkte von M auf einer vertikalen oder horizontalen Geraden liegen. Der kd-Baum von M ist ein binärer Baum, wobei jeder Knoten einen Punkt von M speichert. Sei p der Punkt in M mit mittlerer x-Koordinate und seien $M_l=\{q \text{ in } M/q_x<p_x\}$ und $M_r=\{q \text{ in } M/q_x>p_x\}$. Weiters seien p^l und p^r die Punkte in M_l und M_r mit jeweils mittleren y-Koordinate und seien $M_{l,u}=\{q \text{ in } M_l/q_y<p_y^l\}$, $M_{l,o}=\{q \text{ in } M_l/q_y>p_y^l\}$, $M_{r,u}=\{q \text{ in } M_r/q_y<p_y^r\}$ und $M_{r,o}=\{q \text{ in } M_r/q_y>p_y\}$.

Dann enthält die Wurzel W des kd-Baumes von M p. Der linke Sohn l(W) von W enthält p^l, den kd-Baum von $M_{l,u}$ als linken Teilbaum und den kd-Baum von $M_{l,o}$ als rechten Teilbaum. Analog enthält der rechte Sohn r(W) von W p^r, den kd-Baum von $M_{r,u}$ als linken Teilbaum und der kd-Baum von $M_{r,o}$ als rechten Teilbaum. Abbildung 3-1 zeigt eine Punktmenge und den dazugehörigen kd-Baum.

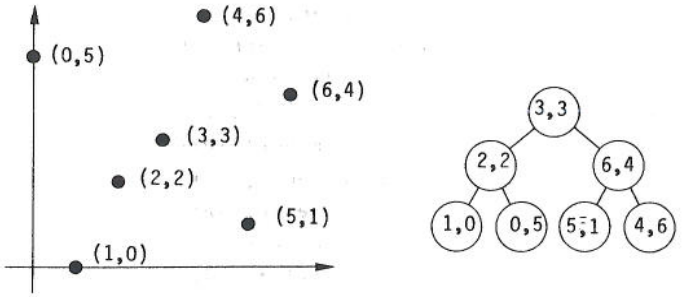

Abbildung 3-1. kd-Baum in zwei Dimensionen.

Unter Verwendung von mehr als O(n) Speicherplatz können Datenstrukturen mit besserer Suchzeit angegeben werden. Wichtige Ergebnisse in dieser Richtung lieferten [Bentley 80a] und [Bentley 80b], bis schließlich [Willard 82] die bis zum heutigen Zeitpunkt effizientesten dieser Strukturen beschrieb. Alle in diesen Arbeiten beschriebenen Datenstrukturen sind Variationen des sogenannten Bereich Baumes.

Sei wiederum M eine Menge von n Punkten in der Ebene mit keinen zweien auf einer vertikalen oder horizontalen Geraden. Falls n=1, so besteht der Bereich Baum von M aus einem Knoten, der diesen Punkt speichert. Andernfalls sei p der Punkt in M mit mittlerer x-Koordinate und seien $M_l=\{q \text{ in } M / q_x \leq p_x\}$ und $M_r=\{q \text{ in } M / q_x > p_x\}$. Die Wurzel des Bereich Baumes von M enthält p_x und das nach den y-Koordinaten sortierte lineare Feld von Punkten in M. Der linke Teilbaum von W ist der Bereich Baum von M_l, der rechte Teilbaum von W ist der Bereich Baum von M_r. Abbildung 3-2 zeigt den Bereich Baum für die Punktmenge in Abbildung 3-1.

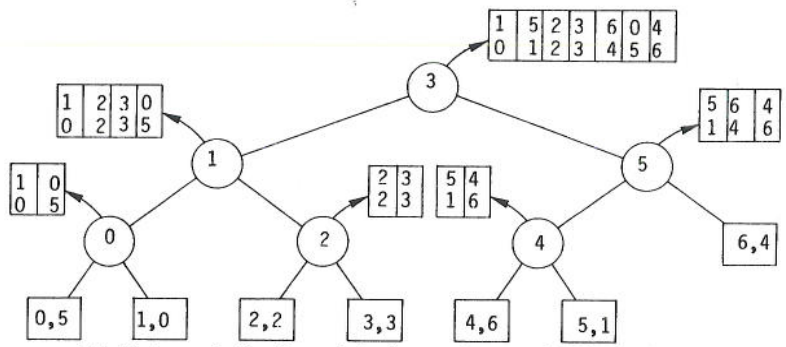

Abbildung 3-2. Bereich Baum in zwei Dimensionen.

Der Bereich Baum für n Punkte in zwei Dimensionen benötigt O(nlogn) Speicher und ermöglicht es, eine Bereichanfrage in $O(\log^2 n+t)$ Zeit zu beantworten. Zusätzliche Zeiger verbessern die Suchzeit auf O(logn+t), siehe [Willard 82].

Ein Problem, verwandt zum Bereich Suchen, beschäftigt sich mit Rechtecken in der Ebene. (Ein Rechteck ist ebenso definiert wie ein Bereich.) Gegeben ist eine Menge M von n Rechtecken und zu bestimmen sind alle Paare von sich schneidenden Rechtecken in M. Lösungen für dieses Problem sind relevant für das Entwerfen von VLSI chips [Bentley 80c] und für das Bestimmen von sich schneidenden Paaren allgemeiner geometrischer Objekte [Edelsbrunner 82]. Eine Lösung, optimal im Speicher-u. Zeitaufwand ergibt sich durch Anwendung des sogenannten Intervall Baumes, der unabhängig voneinander in [Mc Creight 80] und [Edelsbrunner 80] entwickelt wurde.

Sei M eine Menge von n Intervallen auf der reellen Geraden. Der Einfachheit halber seien keine zwei Endpunkte gleich. Sei w ein Wert ungleich jedem Endpunkt, sodaß je n Endpunkte links und rechts von w liegen. Weiters seien M_l, M_m, bzw. M_r die Menge von Intervallen in M, die links von w liegen, w enthalten, bzw. rechts von w liegen. Die Wurzel W des Intervall Baumes von M enthält w, den Intervall Baum von M_l als linken Teilbaum und den Intervall Baum von M_r als rechten Teilbaum. Zusätzlich sind die aufsteigend sortierte lineare Liste der linken Endpunkte und die absteigend sortierte lineare Liste der rechten

Endpunkte der Intervalle in M_m W zugeordnet. Abbildung 3-3 zeigt den Intervall Baum für fünf Intervalle.

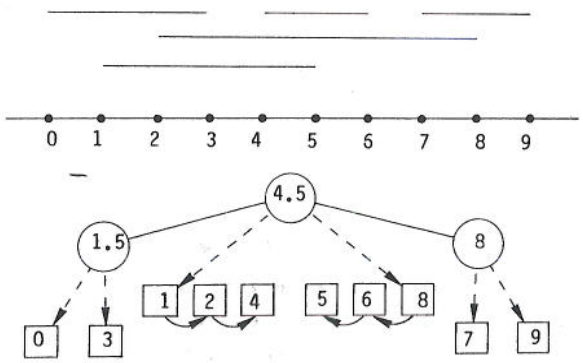

Abbildung 3-3. Der Intervall Baum.

Die Signifkanz des Intervall Baumes besteht in der Unterbringung von n Intervallen in O(n) Speicher, sodaß für ein neues Intervall q die t Intervalle in M, die q schneiden, in O(log n+t) Zeit ausgedruckt werden können.

Wie mit Hilfe des Intervall Baumes das Ausgangsproblem (Bestimmen von schneidenden Rechteckspaaren) gelöst werden kann, soll nun erklärt werden. Die grundlegende Idee liefert die sogenannte plane-sweep Technik: Die Ebene wird gedanklich mit einer vertikalen Geraden V von links nach rechts durchwandert. Mit V wird ein Intervall Baum B mitgeführt, der die y-Intervalle jener Rechtecke enthält, die V schneiden. Stößt V an ein neues Rechteck R, so werden alle Intervalle in B bestimmt, die das y-Intervall von R schneiden. Alle zugehörigen Rechtecke schneiden R und werden zusammen mit R ausgegeben. Außerdem wird das y-Intervall von R in B aufgenommen. Wenn V ein Rechteck R verläßt, so wird das y-Intervall von R aus B entfernt. Die Modifikationen, die nötig sind, damit Intervalle in O(logn) Zeit in einem Intervall Baum für n Intervalle eingefügt oder entfernt werden können, sind in [Edelsbrunner 80b] beschrieben. Als Resultat ergibt sich ein Algorithmus, der die t schneidenden Paare von n Rechtecken in der Ebene in O(nlog n+t) Zeit und O(n) Speicher ausgibt.

3.2 Distanzprobleme

Ein weiterer Typ von Suchproblem, der in Datenbanken auftritt, sucht nach Daten, die einem gegebenen Datum in gewissem Sinn ähnlich sind. Die Ähnlichkeit wird durch die Distanz nach einer definierten Distanzfunktion ausgedrückt. Für Punkte in der Ebene als Daten bietet sich die euklidische Entfernung als natürliches Distanzmaß an. Auch diese Probleme wurden bereits in [Knuth 73] erwähnt. Die bekannteste Version dieser Probleme nennt sich Postamt Problem und fragt für einen Suchpunkt q in der euklidischen Ebene nach dem nächsten Punkt einer vorgegebenen Menge M von n Punkten, siehe [Maurer 79a].

Ein eleganter Zugang zu diesem Problem wurde in [Shamos 75a] beschrieben. Für M wird das Voronoi Diagramm V(M) konstruiert, das die Ebene in n konvexe nicht überlappende Polygone zerlegt. Für einen Suchpunkt q wird der nächste Punkt in M durch Finden des Polygons von V(M), das q enthält, bestimmt. Damit der zweite Schritt korrekt ist, muß V(M) jedem Punkt p in M ein Polygon P(p) so zuordnen, daß p genau dann nächster Punkt von q ist, wenn q in P(p) liegt. Diese Eigenschaft legt das Voronoi Diagramm eindeutig fest. Abbildung 3-4 zeigt das Voronoi Diagramm für die Punktemenge aus Abbildung 3-1.

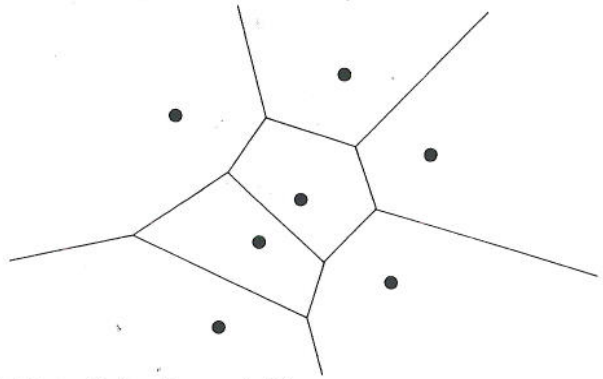

Abbildung 3-4. Voronoi-Diagramm

Hinter der Idee das Voronoi Diagramm für das Postamt Problem einzusetzen steckt eine allgemeine Methode, die der Locus-approach genannt wird: Für jede mögliche Antwort des Suchproblems wird der geometrische Ort aller Suchpunkte berechnet, die zu dieser Antwort führen.

Ein optimaler Algorithmus der V(M) in O(nlogn) Zeit aufbaut, ist in [Shamos 75b] beschrieben. Man beachte, daß damit das Postamt Problem noch nicht gelöst ist. Es bleibt die Aufgabe V(M) so zu speichern, daß für einen Suchpunkt q das Polygon, das q enthält, rasch gefunden werden kann.

Allgemein wollen wir nun das Punkt Lokalisierungs Problem behandeln, das von einem ebenen Graphen G mit m geraden Kanten ausgeht. G soll so gespeichert werden, daß für einen Suchpunkt q mit geringem Aufwand das Polygon P durch G definiert bestimmt werden kann, das q enthält. Man beachte, daß V(M) ein ebener Graph mit $m<3n$ ist. Die erste Lösung wurde in [Dobkin 76] entwickelt. Sie benötigt $O(m^2)$ Speicher und erlaubt das Lokalisieren von q in O(logm) Zeit. Bessere Datenstrukturen wurden in [Lee 77b] (O(m) Speicher und $O(\log^2 m)$ Suchzeit) und in [Preparata 81] (O(mlogm) Speicher und O(logm) Suchzeit) beschrieben.

Eine optimale Lösung, die bei O(m) Speicher O(logm) Suchzeit ermöglicht, existiert seit [Kirkpatrick 81]. Sie geht von einer Triangulierung aus, was keine Einschränkung bedeutet, weil jeder ebene Graph mit geraden Kanten trianguliert werden kann. Um eine vollständige Triangulierung zu erhalten, wird außerdem ein Dreieck über den Graphen gelegt, das alle Knoten enthält. Existieren unbeschränkte Polygone wie beim Voronoi Diagramm, so werden ihre Teile außerhalb des neuen Dreiecks separat behandelt. Da sie total geordnet sind, braucht kein signifikanter Aufwand dafür verwendet werden. Abbildung 3-5 zeigt eine Triangulierung des Voronoi Diagramms aus Abbildung 3-4 .

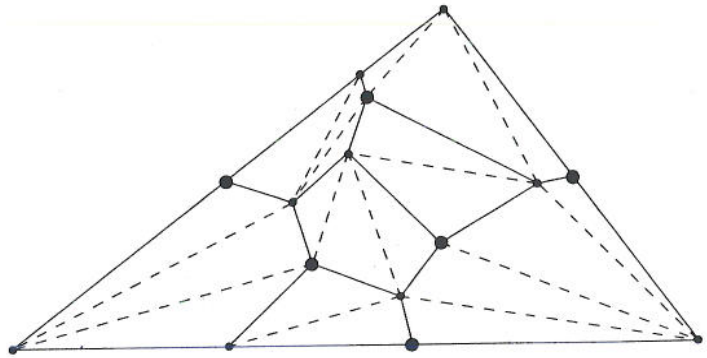

Abbildung 3-5. Triangulierung.

In einem Schritt wird von der Triangulierung, nennen wir sie T_o, eine gröbere Triangulierung T_1 durch folgende Schritte hergeleitet:

(1) Auswahl einer Knotenmenge K in T_o.

(2) Entfernen der Knoten in K samt den inzidenten Kanten aus T_o.

(3) Triangulieren der in (2) entstandenen Polygone.

Die Bedingungen, die bei der Auswahl von K erfüllt werden, sind: (i) keine zwei Knoten in K sind durch eine Kante verbunden, (ii) jeder Knoten in K ist mit höchstens c_1 Kanten inzident (für ein konstantes c_1) und (iii) K enthält zumindest $c_2 n$ der n Knoten von T_o (für ein konstantes c_2). Es kann gezeigt werden, daß c_1 und c_2 existieren, sodaß K die drei Bedingungen erfüllt (z.B. $c_1=11$, $c_2=1/24$).

Durch wiederholtes Vergröbern der Triangulierung ergibt sich eine Hierarchie von Triangulierungen $T_o, T_1 \ldots T_h$, wobei T_h konstant viele Dreiecke enthält. Wegen Bedingung (iii) gilt $h=O(\log n)$ und durch die Bedingungen (i) und (ii) ist garantiert, daß jedes Dreieck in T_i höchstens c_1 Dreiecke in T_{i-1} überlappt, für $1 \le i \le h$. Jedem Dreieck in T_i wird die Menge der Dreiecke in T_{i-1}, die es überlappt, zugeordnet. Für die Hierarchie und die Zuordnungen zwischen zwei in der Hierarchie benachbarten Triangulierungen genügt O(n)

Speicher. Außerdem kann das Dreieck in T_o (der Ausgangstriangulierung), das einen Suchpunkt q enthält, in O(logn) Zeit durch den folgenden Algorithmus bestimmt werden:

> Sei KANDIDATEN die Menge der Dreiecke in T_h und sei D das Dreieck in KANDIDATEN, das q enthält. Sei i=h-1.
>
> Solange i≥0 gilt wird folgendes ausgeführt: Die Menge der Dreiecke in T_i, die D überlappen wird berechnet und KANDIDATEN genannt. Das neue Dreieck D ist jenes Dreieck in KANDIDATEN, daß q enthält. Zuletzt wird i um eins verringert.
>
> Das berechnete Dreieck D ist das gesuchte und wird ausgegeben.

Die gemeinsame Verwendung von Voronoi Diagramm und der Hierarchie von Triangulierungen ergibt eine Datenstruktur, die das Postamt Problem für n Punkte mit O(n) Speicher und O(logn) Suchzeit löst.

Der Anwendungsbereich des Voronoi Diagramms ist mit der Lösung des Postamt Problems in keiner Weise erschöpft. Viele wichtige Probleme, wie das Bestimmen des nächsten Paares in M oder das Konstruieren eines minimalen aufspannenden Baumes, lassen sich mit Hilfe von V(M) lösen. Dazu wird die Eigenschaft ausgenutzt, daß diese Probleme Graphen für M als Knotenmenge induzieren, die Teilgraphen des dualen Graphen von V(M) sind. Dieser duale Graph, genannt die Delaunay Triangulierung DT(M) von M, verbindet zwei Punkte p_1 und p_2 in M durch eine gerade Kante genau dann, wenn die durch V(M) induzierten Polygone $P(p_1)$ und $P(p_2)$ eine Kante gemeinsam haben; siehe Abbildung 3-6.

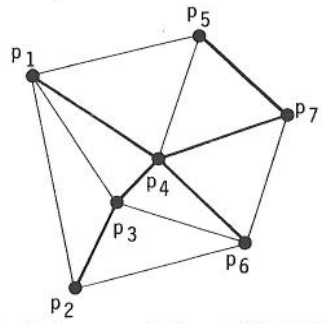

Abbildung 3-6. Delaunay Triangulierung.

Um das nächste Paar von Punkten in M zu bestimmen, wird im ersten Schritt DT(M) konstruiert und im zweiten Schritt das nächste unter den Paaren, die durch Kanten in DT(M) definiert sind, eruiert. (In der Punktemenge aus Abbildung 3-6 ist (p_3,p_4) das nächste Paar.) Der Aufwand beträgt O(nlogn) Zeit für DT(M) und zusätzlich O(n) Zeit, um jedes Paar, das durch eine Kante verbunden ist, zu untersuchen. Mit der prizipiell unveränderten Methode kann auch für jeden Punkt p in M sein nächster Punkt in M bestimmt werden.

Ein für viele Anwendungen relevantes Problem ist das Konstruieren eines minimalen aufspannenden Baumes MST(M) von M. Ein MST(M) ist ein Teilgraph aus MxM, sodaß zwischen je zwei Punkten genau ein Weg in MST(M) besteht, und die Summe der Kantenlängen das Minimum annimmt. Ein minimaler aufspannender Baum der sieben Punkte,dargestellt in Abbildung 3-6, besteht aus den Kanten (p_1,p_4), (p_2,p_3), (p_3,p_4), (p_4,p_7), (p_4,p_6) und (p_5,p_7). Weil jeder MST(M) Teilgraph von DT(M) ist, siehe [Shamos 75b], kann durch sorgfältige Implementierung des Algorithmus von [Prim 57] mit zusätzlichem Aufwand von O(nlogn) Zeit ein MST(M) aufgebaut werden.

3.3 Konvexe Hüllen

Ein weiteres Problem, das durch Aufbau des Voronoi Diagramms gelöst werden kann, ist die Bestimmung der konvexen Hülle KH(M) von M. Die konvexe Hülle von M ist das kleinste konvexe Polygon, das alle Punkte von M enthält. Oblicherweise wird KH(M) durch die Reihenfolge der Punkte von M, die auf dem Rand von KH(M) liegen, gespeichert. Diese Reihenfolge ist für die Abbildung 3-6 dargestellte Punktemenge p_1,p_2,p_6,p_7,p_5. Der Rand von KH(M) ist Teilgraph von DT(M) und kann in O(N) Zeit von DT(M) abgeleitet werden.

Es existieren jedoch auch programmiertechnisch wesentlich einfachere Algorithmen die KH(M) in O(nlog) Zeit berechnen. Der historisch gesehen erste dieser Algorthmen wurde von [Grahäm 72] angegeben. Der Einfachheit halber nehmen wir

an, daß keine drei Punkte von M auf einer Geraden liegen. Im ersten Schritt wird ein sternförmiges Polygon bestimmt, dessen Eckpunkte die Punkte von M bilden. Zu diesem Zweck wird ein neuer Punkt Z innerhalb von KH(M) berechnet (etwa der Schwerpunkt irgend eines Dreiecks bestimmt durch drei Punkte aus M). Jeder Punkt p von M definiert einen Winkel w(p) zwischen dem horizontalen Halbstrahl, der Z nach rechts verläßt und dem Halbstrahl, der von Z ausgeht und p enthält. Die bezüglich der Winkel w(p) sortierte Reihenfolge der Punkte in M definiert ein sternförmiges Polygon, siehe Abbildung 3-7.

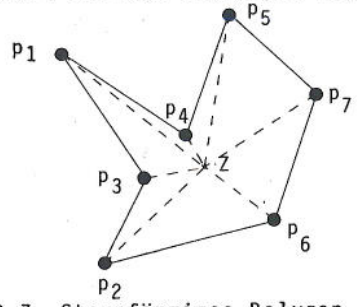

Abbildung 3-7. Sternförmiges Polygon.

Im zweiten Schritt werden durch den folgenden Algorithmus die Konkavitäten entfernt. Wir bezeichnen mit NACH(p), bzw. VOR(p), den Punkt, der im Gegenuhrzeigersinn unmittelbar nach, bzw. vor, p liegt. Z.B. gilt P_1=NACH(p_4)=VOR(p_3) im Polygon der Abbildung 3-7. Weiters sei IW(p) der Innenwinkel von p, definiert durch die Kanten (VOR(p),p) und (p,NACH (p)).

Sei p (anfangs $p=p_1$) der laufende Punkt.
Der folgende Block von Anweisungen
wird 2n mal ausgeführt:

Fall 1: IW(p)<180^o. Dann wird
p auf NACH(p) gesetzt.
Fall 2: IW(p)>180^o. Dann wird
q=p und p=VOR(q) gesetzt.
Der Punkt q wird aus dem Polygon
entfernt.

In gewissem Sinn wird bei diesem Algorithmus das Berechnen der konvexen Hülle auf das Sortieren von Zahlen zurückgeführt, weil das Sortieren der Winkel den aufwendigsten Teil ausmacht. Ein im prinzipiellen Ansatz verschiedener Algorithmus wurde in [Jarvis 73] vorgestellt. Vereinfachend setzen wir voraus, daß keine drei Punkte auf einer Geraden liegen, und daß außerdem keine zwei Punkte auf einer horizontalen Geraden liegen.

Schritt 1: Sei p der Punkt p_o in M mit
minimaler y-Koordinate und sei W=0.

Schritt 2: Bestimme den ersten Punkt q,
auf den der in p verankerte Halbstrahl H
stößt. Zu diesem Zweck sei W der Winkel
zwischen H und dem horizontalen Halb-
strahl, der p nach rechts verläßt. Ausgehend von der durch W festgelegten Position
wird H im Gegenuhrzeigersinn um p geschwenkt.
q ist der nächste Punkt auf der konvexen
Hülle, deshalb wird p durch q und W durch
den neuen Winkel ersetzt. Falls $p \neq p_o$ wird
Schritt 2 wiederholt. Andernfalls ist die
konvexe Hülle fertig.

Bezeichnet h die Anzahl der Punkte von M auf dem Rand von KH(M), dann benötigt der dargestellte Algorthmus O(hn) Zeit, weil pro Ausführung von Schrit 2 O(n) Zeit nötig ist. Im schlechtesten Fall gilt h=n, und somit benötigt der Algorithmus $O(n^2)$ Zeit im schlechtesten Fall. Der Algorithmus ist ob seiner Einfachheit und der Tatsache, daß h für viele Verteilungen in O(logn) ist, trotzdem von praktischem Interesse.

Aus der umfangreichen Literatur zum Thema "konvexe Hülle" wollen wir zwei weitere Arbeiten hervorheben: [Preparata 77] beschrieb Algorithmen basierend auf dem Divide-and-Conquer Prinzip, die die konvexe Hülle für n Punkte in zwei oder drei Dimensionen in O(nlogn) Zeit

aufbauen. [Seidel 82] entwickelte Algorithmen, die für beliebige Dimensionen d funktionieren und für geradzahlige d optimal sind, d.h. in $O(n\log n+n^{d/2})$ Zeit arbeiten.

3.4 Dynamisierungsmethoden

Bis dato haben wir nur statische Datenstrukturen besprochen, d.h. Datenstrukturen für unveränderliche Mengen M von Punkten oder Intervallen. Für viele Anwendungen ist es jedoch notwendig, gegebenenfalls Objekte in die Menge einzufügen oder zu entfernen. Wenn eine Datenstruktur das Einfügen und Entfernen von Objekten ohne großen Aufwand erlaubt, so nennen wir sie dynamisch. Ein AVL-Baum ist z.B. eine dynamische Datenstruktur, die die Sortierung von eindimensionalen Werten aufrechterhält.

In allen bekannten Fällen sind dynamische Strukturen den entsprechenden statischen sehr ähnlich und können durch gewisse allgemein verwendbare Transformationen aus den letzteren abgeleitet werden. Diese Transformationen, die statische Datenstrukturen dynamisieren, werden Dynamisierungsmethoden genannt. Es gibt zwei prinzipiell voneinander verschiedene Dynamisierungsmethoden: Die eine kann durch den Einsatz und die Adaption von balanzierten Bäumen charakterisiert werden. Die andere hält ein System von Datenstrukturen aufrecht, wobei jede dieser Datenstrukturen nach dem Vorbild der zu dynamisierenden Datenstruktur errichtet wird. Die erste Methode wird anhand des Bereich Baumes, die zweite anhand des Voronoi Diagramms, illustriert.

Zunächst der Bereich Baum. Erinnern wir uns an die Struktur dieses Baumes: Es ist ein binärer Baum (nun genannt die Primärstruktur), der die bezüglich der x-Koordinaten sortierten Punkte in seinen Blättern speichert. Außerdem ist jedem inneren Knoten k ein lineares Feld D (genannt die Sekundärstruktur von k) zugeordnet, das die Punkte in den Blättern des Teilbaumes mit Wurzel k sortiert bezüglich y-Koordinaten enthält.

Der erste Schritt der Dynamisierung ersetzt jedes lineare Feld durch ein Wörterbuch. Wird nun ein Punkt p eingefügt (oder entfernt), so wird p (unter der Annahme, daß keine Änderung in der Primärstruktur stattfindet) in (aus) O(logn) Sekundärstrukturen in jeweils O(logn) Zeit eingefügt (entfernt). Der Aufwand für diese Operationen beträgt $O(\log^2 n)$ Zeit. Notwendige Änderungen in der Primärstruktur sind zu diesem Zeitpunkt noch nicht berücksichtigt. Im zweiten Schritt wird der optimal balanzierte Baum, der die Primärstruktur darstellt, durch einen gewichtsbalanzierten Baum ersetzt, siehe [Nievergelt 73]. An dieser Stelle ist es notwendig, genauer auf gewichtsbalanzierte Bäume und ihre Umstrukturierungsmechanismen einzugehen.

Sei B ein binärer Baum, wobei jeder Knoten k entweder zwei Söhne hat oder ein Blatt ist. Die Anzahl G(k) der Blätter im Teilbaum mit Wurzel k wird das Gewicht von k genannt. Sei k ein innerer Knoten und k_1 der linke Sohn von k, dann heißt $b(k)=G(k_1)/G(k)$ die Balance von k. B ist α-balanziert für α in $(0, 1/2]$, wenn für jeden inneren Knoten k seine Balance in $[\alpha,1-\alpha]$ ist. Für $\alpha=1-\sqrt{2}/2$ konnten [Nievergelt 73] zeigen, daß die Rotation als Umstrukturierung ausreicht. Wenn ein Knoten k infolge einer Einfügung oder Entfernung außer Balance gerät, dann kann durch ein- oder zweimalige Anwendung der Rotation b(k) in das erforderliche Intervall zurückgezwungen werden. Abbildung 3-8 zeigt die Rotation von k nach rechts. Die Rotation nach links erhält man durch Spiegelung des Bildes an einer vertikalen Geraden.

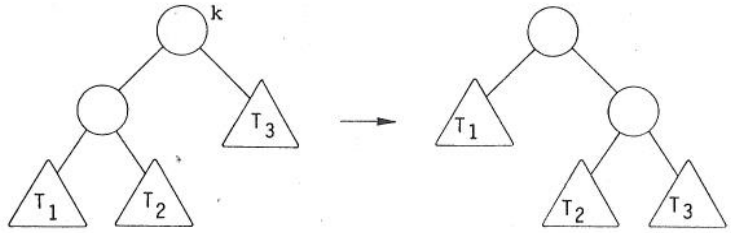

Abbildung 3-8. Rotation nach rechts.

Eine Rotation von k nach rechts führt zum Ziel, wenn $b(k)>1-\alpha$ und für den linken Sohn k_1 von k $b(k_1)>(1-2\alpha)/(1-\alpha)$ gilt. Für $b(k_1)\leq(1-2\alpha)/(1-\alpha)$ wird zunächst k_1 nach links, dann k nach rechts rotiert.

Die unangenehme Nebenwirkung der Anwendung einer Rotation auf die Primärstruktur des Bereich Baumes (die nun α-balanziert ist) besteht darin, daß für einen Sohn des rotierten Knotens eine völlig neue Sekundärstruktur erstellt werden muß. Besonders nachteilig wirkt sich diese Tatsache aus, wenn der rotierte Knoten nahe der Wurzel liegt. [Willard 78] zeigte, daß Knoten mit hohem Gewicht selten rotiert werden, und daher $O(n\log^2 n)$ Zeit genügt, um n Einfügungen und Entfernungen bei einem anfangs leeren dynamischen Bereichsbaum auszuführen. In weiterer Folge gab er komplizierte Mechanismen an, die garantierten, daß eine Einfügung oder Entfernung in $O(\log^2 n)$ Zeit ausgeführt werden kann. Zu beachten ist, daß die Suchzeit für einen Bereich durch die Dynamisierung asymptotisch nicht beeinträchtigt wird.

Nun das Voronoi Diagramm. Es können Beispiele von n Punkten konstruiert werden, sodaß die für das Einfügen oder Entfernen eines Punktes benötigte Zeit zumindest proportional zu n ist. Das ist immer dann der Fall, wenn die Änderung einen Punkt betrifft, dessen Polygon extrem viele Kanten aufweist. Um solche Situationen zu vermeiden, haben [Maurer 79b] vorgeschlagen, die Menge M von n Punkten in ungefähr $\sqrt{n}$ disjunkte Teilmengen zu je ungefähr $\sqrt{n}$ Punkte aufzuteilen. Sei $M=M_1 \dot\cup M_2 \dot\cup \ldots \dot\cup M_t$ eine solche Aufteilung. Für jede Menge M_i wird das Voronoi Diagramm $V(M_i)$ aufgebaut. Wenn es um die Lösung des Postamt Problems für M geht, wird jedes Diagramm als Hierarchie von Triangulierungen gespeichert.

Sei q ein Suchpunkt, für den der nächste Nachbar in M bestimmt werden soll. Zu diesem Zweck wird der nächste Punkt p^i in M_i bestimmt, für $1\leq i\leq t$. Der nächste Punkt in M ist der nächste der Punkte p^i, deren Entfernungen zu q explizit verglichen werden. Eine Suche benötigt somit $O(\sqrt{n}\log n)$ Zeit.

Diese Verschlechterung der Suchzeit wird durch die Verbesserung des Aufwandes, um einen Punkt p in M einzufügen oder zu entfernen, kompensiert. Im ersten Fall wird p in irgend eine (am besten in die kleinste) der Mengen M eingefügt. Beim Entfernen wird p aus jener Menge M_i entfernt, die p enthält. Die Adjustierung der Menge M_i geschieht in beiden Fällen durch völliges Neuaufbauen von $V(M_i)$ und $H(M_i)$. Einige Details sind zu diesen Manipulationen anzumerken.

(1) Um beim Entfernen von p die Menge M_i zu bestimmen, die p enthält, kann beispielsweise der nächste Punkt zu p in jeder Menge gesucht werden. Für andere Datenstrukturen, die kein Suchen eines Objektes ermöglichen, jedoch trotzdem mit der beschriebenen Methode dynamisiert werden, wird das Auffinden von p durch ein Wörterbuch ermöglicht. Dieses enthält alle Objekte und für jedes Objekt die Menge, die es enthält.

(2) Durch oftmaliges Einfügen bzw. Entfernen kann das System in Unordnung gebracht werden, da sich die laufende Anzahl n von Punkten stark ändern kann. Außerdem ist es möglich, daß eine Menge M_i durch wiederholtes Entfernen ab einem gewissen Zeitpunkt weit von seiner Soll-Größe entfernt ist. Wie diese unerwünschten Situationen ohne großen Aufwand vermieden werden, ist in [Van Leeuwen 80] beschrieben.

In der für das Voronoi Diagramm beschriebenen Dynamisierungsmethode wird die Punktemenge in kleinere, jedoch untereinander ungefähr gleich große Mengen aufgeteilt. Jede dieser Teilmengen wird separat strukturiert. In vielen Fällen (besonders wenn das Entfernen aus der statischen Struktur leicht, das Einfügen jedoch schwer ist), bewährt sich die Aufteilung der Menge in verschieden große Teile. Eingefügt wird nur in die kleinste Menge; größere Mengen entstehen durch Vereinigung von kleinen Mengen. Erstmals entdeckt

wurde diese Methode von [Bentley 79]. Eine kombinierte Verwendung aller bekannten nützlichen Mechanismen kann in [Overmars 81] gefunden werden.

4. Neuere Resultate

Dieses Kapitel ist letzten Ergebnissen über geometrische Algorithmen und Datenstrukturen gewidmet. Im speziellen werden drei Lösungen von geometrischen Problemen vorgestellt. Diese sind so gewählt, daß sie weitgehend unabhängig voneinander sind und damit einen kleinen Querschnitt des heutigen Schaffens repräsentieren. Darüber hinaus werden Richtungen für zukünftige Forschung und spezielle offene Probleme hervorgehoben, wenn sie sich aus dem Zusammenhang ergeben.

4.1 Halbebenen Suchproblem

Gegeben ist eine Menge von n Punkten, die so gespeichert werden soll, daß für jede Halbebene h die Anzahl von Punkten aus M in h rasch bestimmt werden kann. Dieses Suchproblem ist mit dem Bereich Suchproblem aus Kapitel 3.1 vergleichbar. Der Unterschied liegt in der Art der Suchobjekte und in der verlangten Antwort: Zuvor wurde nach den Punkten in einem orthogonalen Rechteck gefragt. Jetzt ist die Anzahl der Punkte in einer Halbebene gesucht. Es zeigt sich das im ersten Augenblick vielleicht überraschende Ergebnis, daß das Halbebenen Suchproblem wesentlich komplexer ist, als das Bereich Suchproblem. Argumente für diese Behauptung wurden in [Fredman 80] dargelegt. Aus diesem Grund ist die Untersuchung von sogenannten Halbebenen Schätzproblemen interessant, die Vergröberungen des Halbebenen Suchproblems darstellen. Die einfachste Version der Halbebenen Schätzprobleme fragt für eine Halbebene h, ob h zumindest $\lceil n/2 \rceil$ Punkten von M enthält. Im folgenden wird dieses spezielle Problem kurz das Schätzproblem genannt.

Um eine anschauliche Darstellung einer Datenstruktur von [Edelsbrunner 82b], die das Schätzproblem löst, zu gestatten, verwenden wir eine geometrische Transformation D. Diese ordnet jedem Punkt $p=(p_x,p_y)$ in M die Gerade D(p) zu

deren Punkte (x,y) die Gleichung $y=p_x x+p_y$ erfüllen. Der Einfachheit halber nehmen wir an, daß keine drei Punkte aus M auf einer Geraden liegen. Daraus ergibt sich auch, daß sich keine drei Geraden aus D(M) in einem Punkt treffen. Außerdem wird D verwendet, um eine Gerade g, deren Punkte (x,y) die Gleichung $y=g_1 x+g_2$ erfüllen, in den Punkt $D(g)=(-g_1,g_2)$ zu transformieren. Die Nützlichkeit von D leitet sich aus folgender Eigenschaft ab:

Ist p ein Punkt und g eine Gerade, dann
liegt p genau dann über (auf, unter) g,
wenn D(g) unter (auf, über) D(p) liegt.

Abbildung 4-1 zeigt die Punktmenge aus Abbildung 3-1 samt einer Geraden unddas Ergebnis der Transformation D, auf diese Konfiguration angewendet.

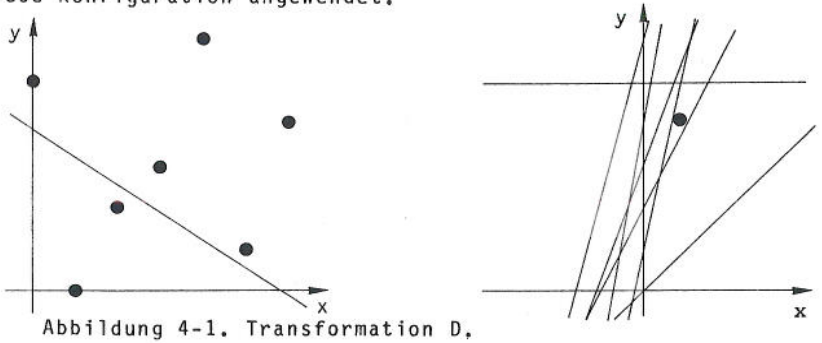

Abbildung 4-1. Transformation D.

Beschränken wir uns der Einfachheit halber auf Halbebene h, die oben durch eine Gerade g begrenzt sind. Jeder Punkt p in h korrespondiert mit einer Geraden D(p), sodaß D(g) über D(p) liegt. Diese Überlegung legt nahe, den Polygonzug P im Geraden-Arrangement D(M) zu berechnen, sodaß genau dann zumindest $\lceil n/2 \rceil$ Geraden von D(M) unter einem Punkt p liegen oder p enthalten, wenn p über P oder auf P liegt. Einfache Überlegungen ergeben, daß P jede vertikale Gerade genau einmal schneidet: Daher ist es berechtigt, davon zu sprechen, daß p über oder unter P liegt.

Abbildung 4-2 zeigt den Polygonzug für sieben Geraden. Der Polygonzug P für D(M) besteht aus jenen Kanten, für

deren innere Punkte gilt, daß genau $\lceil n/2 \rceil$ Geraden von D(M) darunter liegen oder sie enthalten. Weil P monoton in x ist, kann für einen Suchpunkt D(g) durch binäres Suchen in $O(\log|P|)$ Zeit entschieden werden, ob D(g) über, auf oder unter P liegt. (|P| bezeichnet die Anzahl der Kanten von P.)

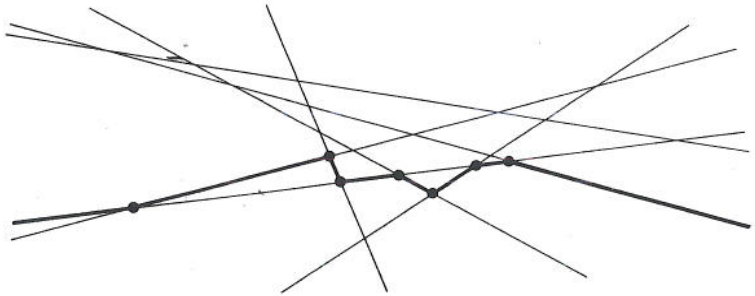

Abbildung 4-2. Polygonzug für Gerade-Arrangement.

Der Speicheraufwand und die Zeit für den Aufbau der Datenstruktur, die die Kanten von P in linearer Folge speichert, ist abhängig von |P|. Der in [Edelsbrunner 82b] dargestellte Algorithmus benötigt O(|P|) Speicher und baut P in $O(|P|\log^2 n + n\log n)$ Zeit auf. Die interessandte Frage, wie groß |P| für n Gerade werden kann, ist derzeit nur unbefriedigend beantwortet. [Edelsbrunner 80a] zeigt, daß Beispiele von n Geraden existieren, sodaß |P| in $\Omega(n\log n)$ liegt. Als obere Schranke wurde $O(n^{3/2})$ hergeleitet. Varianten des Schätzproblems und Methoden zur Reduktion des Speicheraufwandes sind in [Edelsbrunner 82b] behandelt.

Wichtig erscheint bei der beschriebenen Lösung die Verwendung von geometrischen Transformationen als Hilfsmittel zur algorithmischen Behandlung. Dieses Hilfsmittel wurde das erste Mal von [Brown 80] in größerem Rahmen vorgestellt und erweist sich immer wieder als hilfreich. Die enge Beziehung zwischen mehrdimensionalen Algorithmen bzw. Datenstrukturen und elementargeometrischer Fragen erscheint uns als weiterer wichtiger Aspekt. Einen Einstieg in den letzteren Problemkreis erleichtern die umfassenden und tiefreichenden Werke

[Grünbaum 67] und [Grünbaum 72].

4.2 Gewichtete Voronoi Diagramme

Voronoi Diagramme sind bereits in Kapitel 3.2 besprochen worden. Eine intuitive Interpretation des Diagramms für eine Menge M von n Punkten in der Ebene betrachtet das Voronoi Polygon P(p) eines Punktes p aus M als den Einflußbereich von p. Dabei wird implizit angenommen, daß jeder Punkt von M in gewissem Sinn gleich stark ist, Wird jedem Punkt p ein positives Gewicht w(p), das die Stärke des Einflusses von p auf seine Umgebung mißt, zugeordnet, so ist V(M) weit davon entfernt die Einflußstruktur von M widerzuspiegeln. Einen Ausdruck für diese Struktur liefert das sogenannte gewichtete Voronoi Diagramm WV(M) von M. Sei p ein Punkt aus M, a irgend ein Punkt der Ebene, und bezeichne d(a,p) den euklidischen Abstand zwischen a und p. Dann nennen wir $d_w(a,p)=d(a,p)/w(p)$ den gewichteten Abstand von a zu p. In WV(M) wird jedem Punkt p von M als Region R(p) der Ort der Punkte zugeordnet, deren gewichteter nächster Nachbar p ist.

Die Region eines Punktes ist durch Kreisbögen (die im Grenzfall bei gleichgewichteten Punkten zu Geradestücken ausarten) begrenzt, siehe [Aurenhammer 83] und Abbildung 4-3. Das dargestellte Diagramm macht darüberhinaus deutlich, daß Regionen im allgemeinen weder konvex noch zusammenhängend sind. Außerdem sind zusammenhängende Teile einer Region im allgemeinen nicht einfach zusammenhängend.

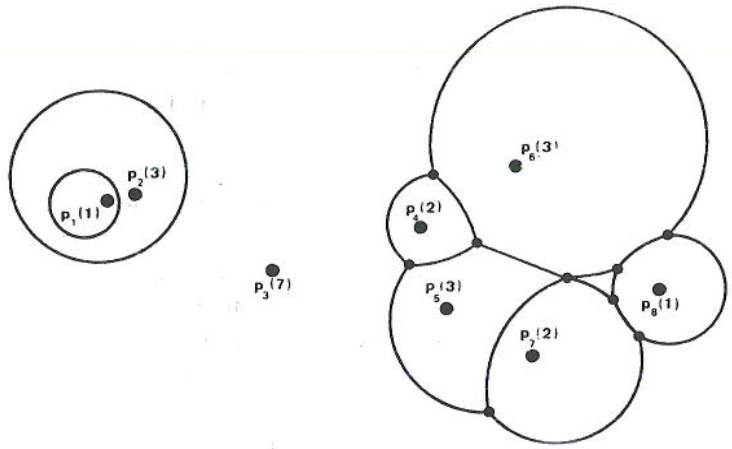

Abbildung 4-3. Gewichtetes Voronoi Diagramm.

Es gibt Beispiele von n-punktigen Mengen M, sodaß WV(M) aus $\Omega(n2)$ Kanten besteht. Man kann zeigen, daß $O(n^2)$ eine obere Schranke für die Anzahl der Knoten, Kanten und Flächen in WV(M) ist. Ein Algorithmus, der WV(M) in $O(n^2)$ Zeit aufbaut, ist in [Aurenhammer 83] beschrieben. Dieser ordnet mittels geometrischer Transformation dem zweidimensionalen Diagramm einen dreidimensionalen Komplex von konvexen Polyedern zu. Das Diagramm ergibt sich aus der Rücktransformation des Schnittes dieses Komplexes mit einer speziellen Kugeloberfläche.

Im folgenden werden die geometrische Transformation und die wichtigsten Stationen des Algorithmus kurz beschrieben: Seien p und q zwei Punkte der Ebene mit Gewichten $w(p)<w(q)$. Der Ort K(p,q) von Punkten a, sodaß $d_w(a,p)<d_w(a,q)$, bildet das Innere eines Kreises, der auch p enthält. Siehe p_1 und p_2 in Abbildung 4-3. Für die Punkte im Inneren des Komplements K(q,p) von K(p,q) leigt q gewichtet näher als p. Sei $M=\{p_1, .. p_n\}$, dann ist $R(p_i)$ gleich dem Durchschnitt aller $K(p_i,p_j)$, für alle $j\neq i$.

Für die Einbettung von WV(M) in drei Dimensionen nehmen wir an, daß die Punkte p_i in der xy-Ebene E des dreidimensionalen Raumes liegen. Ein Punkt I (nicht in E) wird ausgezeichnet und für die Transformation verwendet: Jedem Gebiet

$K(p_i,p_j)$, mit $i \neq j$, wird das Innere oder Äußere $B(p_i,p_j)$ einer Kugeloberfläche zugeordnet, sodaß I auf der Kugeloberfläche liegt und $B(p_i,p_j)$ geschnitten mit E $K(p_i,p_j)$ ergibt. $B(p_i,p_j)$ wird durch Inversion am Zentrum I in einen Halbraum $H(p_i,p_j)$ transformiert. (Die Inversion ordnet jedem Punkt a den Punkt I(a) auf der Geraden durch I und a zu, sodaß $d(I,I(a))=1/d(I,a)$. Der Durchschnitt der $H(p_i,p_j)$, für alle $j \neq i$, ergibt das p_i zugeordnete Polyedern $P(p_i)$. Sobald der Komplex C(M), bestehend aus den Polyedern $P(p_i)$, für $1 \leq i \leq n$, aufgebaut ist, kann WV(M) durch Inversion des Schnittes von C(M) und I(E) gewonnen werden.

Somit reduziert sich der zum Aufbau von WV(M) verwendete Algorithmus auf die Konstruktion der Komplexes C(M). Dieser wird durch sukzessives Einfügen von Punkten (in beliebiger Reihenfolge) aufgebaut. Wir verzichten hier auf weitere Einzelheiten und verweisen auf [Aurenhammer 83].

Die Motivation zur Untersuchung von gewichteten Voronoi Diagrammen stammt aus Gebieten außerhalb der Informatik. Z.B. in der Geographie werden verschiedenste Arten von gewichteten Voronoi Diagrammen (von denen unseres einen Typ darstellt) benötigt. Die Frage, ob das Diagramm für additiv gewichtete Punkte in O(nlogn) Zeit aufgebaut werden kann, ist ein offenes Problem. Dabei bezeichnet d(a,p)-w(p) den Abstand von a zum positiv gewichteten Punkt p. Weiters ist es interessant, die beiden Konzepte der Gewichtung zu kombinieren, oder auch sich bewegende Punkte in der Menge zuzulassen.

4.3 Ein neuer Algorithmus für konvexe Hüllen

Im Kapitel 3.3 sind zwei Algorithmen erläutert worden, die die konvexe Hülle einer Menge M von n Punkten in der Ebene aufbauen. Der Algorithmus von [Graham 72] benötigt dazu O(nlogn) Zeit, jener von [Jarvis 73] O(hn) Zeit, wenn h Punkte von M auf seiner konvexen Hülle liegen. Damit ergab sich die wenig zufriedenstellende Situation, daß es von der

Punktemenge abhängt, welches der beiden Verfahren effizienter arbeitet. Ein Schlußstrich unter dieses Dilemma wurde vor kurzem durch [Kirkpatrick 82] gezogen. Sie entwickelten einen Algorithmus, der beide Verfahren verbessert und KH(M) in O(nlogh) Zeit berechnet.

Zur einfacheren Darstellung nehmen wir an, daß keine drei Punkte von M auf einer Geraden liegen, und daß keine zwei Punkte auf einer vertikalen Geraden liegen. Ohne Beschränkung der Allgemeinheit beschäftigen wir uns nur mit dem Aufbau der oberen konvexen Hülle OH(M) von M. Es stehe eine Prozedur BRÜCKE zur Verfügung, die für M und eine vertikale Gerade V jene Kanten von OH(M) bestimmt, die V schneidet. Der Algorithmus zur Berechnung von KH(M) arbeitet dann wie folgt:

> Eine vertikale Gerade V wird berechnet, die M in zwei Teilmengen M_1 und M_2 von je etwa n/2 Punkten aufteilt.
>
> Durch Anwendung von Prozedur BRÜCKE für M und V wird die Kante (p_i,p_j) (p_i aus M_1 und p_j aus M_2) von OH(M) bestimmt, die V schneidet.
>
> Sei M_1' (bzw. M_2') die Teilmenge von M_1 (bzw. M_2) links (bzw. rechts) von p_i (bzw. p_j). Nun wird der Algorithmus rekursiv zunächst für M_1' und dann für M_2' aufgerufen.

Es kann gezeigt werden, daß OH(M) in O(nlogh) Zeit bestimmt wird, wenn die Prozedur BRÜCKE für n Punkte O(n) Zeit benötigt, siehe [Kirkpatrick 82]. Für Prozedur BRÜCKE wird das Eliminations-Prinzip eingesetzt, das einen proportionalen Anteil der Punkte pro Durchgang eliminiert. Wir sagen, daß eine Gerade g steiler als eine andere Gerade g' ist, wenn die Steigung von g größer als die von g' ist.

Prozedur BROCKE:

Sei M die Menge der Punkte und V die vertikale Gerade, sodaß kein Punkt von M auf V liegt und sich auf beiden Seiten von V Punkte aus M befinden.

Zunächst werden die Punkte von M zu Paaren zusammengefaßt, sodaß $\lfloor n/2 \rfloor$ Geraden entstehen. Sei g die Gerade mit mittlerer Steigung und sei g* die Gerade parllel zu g, sodaß g* zumindest einen Punkt von M enthält, alle anderen aber unter g* liegen.

Fall 1: g* enthält je einen Punkt von M links und rechts von V. Dann enthält g auch die gesuchte Kante.

Fall 2: g^* enthält nur Punkte links von V. Dann ist g^* steiler als die gesuchte Kante und alle linken Punkte von Geraden, die zumindest so steil sind wie g* , werden eliminiert.

Fall 3: g* enthält nur Punkte rechts von V. Dann ist die gesucht Kante steiler als g* und alle rechten Punkte von Geraden, die höchstens so steil sind wie g* , werden eliminiert.

In jedem Durchgang werden ungefähr ein Viertel der noch zu betrachtenden Punkte eliminiert. Der Zeitaufwand pro Durchgang mit n Punkten beträgt O(n), also ist der ingesamte Zeitaufwand in O(n). Als essentieller Bestandteil der Prozedur wird ein Algorithmus verwendet, der die mittlere von n Zahlen bestimmt. Ein Algorithmus, der mit O(n) Zeit auskommt, kann in [Aho 74] gefunden werden.

Als besonders wichtig erscheint uns an diesem Algorithmus für konvexe Hüllen die Verwendung des Eliminations-Prinzips zur algorithmischen Lösung. Eine weitere Anwendung

des Prinzips für lineare Programmierprobleme ist in [Dyer 82] beschrieben.

5. Allgemeine Prinzipien des Programmierens

Dem aufmerksamen Leser wird die Tatsache nicht entgangen sein, daß grundlegende Ideen und Prinzipien wiederholt zu effizienten algorithmischen Lösungen von Problemen geführt haben. Es ist ein primäres Anliegen im Bereich Datenstrukturen, diese Ideen zu erarbeiten und soweit wie möglich zu konkretisieren. Diesem Anliegen ist in den vorangegangenen Kapiteln insofern Rechnung getragen worden, als die verwendeten Prinzipien jeweils kurz herausgestrichen wurden. Dieses Kapitel geht nun explizit auf die einzelnen Prinzipien ein. Es stellt die Charakteristika dieser kurz dar und verweist auf jene Stellen in den vorangegangenen Kapiteln, an denen die Prinzipien zur Anwendung kamen.

5.1 Divide-and-Conquer

Dieses Prinzip kann zur Lösung von Problemen herangezogen werden, die in einem gewissen Sinn zerlegbar sind. In der klassischen Version wird die Datenmenge zunächst in zwei ungefähr gleich große und disjunkte Teilmengen zerlegt. Nachdem die Ergebnisse für beide Teilmengen rekursiv errechnet worden sind, wird das Ergebnis für die gesamte Menge von den beiden Teilergebnissen abgeleitet. Um dieses Prinzip präziser darstellen zu können, sei P ein Problem, M eine Datenmenge und sei P(M) das Ergebnis, nach dem gefragt wird. Sei c eine positive Konstante, die in der Darstellung des Prinzips Verwendung findet. Häufig kann c=1 gesetzt werden.

Fall 1: Enthält M höchstens c Daten, so wird P(M) duch irgend einen trivialen Algorithmus errechnet.

Fall 2: Enthält M mehr als c Daten, dann werden die folgenden Schritte ausgeführt:

Schritt 1: M wird in zwei ungefähr gleich große Teilmengen M_1 und M_2 aufgeteilt.

Schritt 2: $P(M_1)$ und $P(M_2)$ werden rekursiv berechnet.

Schritt 3: P(M) wird von $P(M_1)$ und $P(M_2)$ abgeleitet.

Das Prinzip des Divide-and-Conquer findet z.B. beim Sortieren durch Verschmelzen (siehe Kapitel 2) und beim Aufbau der konvexen Hülle (siehe Kapitel 4.3) Anwendung. An diesen Beispielen werden zwei Varianten des Aufteilens einer Datenmenge sichtbar: Die Menge kann beliebig oder mit Rücksicht auf eine Ordnung der Daten geteilt werden. Der kritischste Teil jedes Algorithmus, der nach dem Divide-and-Conquer Prinzip arbeitet, ist die in Schritt 3 auszuführende Herleitung von P(M) aus $P(M_1)$ und $P(M_2)$. Kann dieser Schritt effizient gelöst werden, dann ergibt sich eine effiziente Gesamtlösung.

5.2 Eliminieren

Sei wiederum P ein Problem, M eine Datenmenge und P(M) das zu errechnende Ergebnis. Sei außerdem c eine passende positive Konstante. Dann kann das Prinzip des systematischen Eliminierens von Daten wie folgt dargestellt werden:

Fall 1: Enthält M höchstens c Daten, so wird P(M) durch irgend einen trivialen Algorithmus errechnet.

Fall 2: Enthält M mehr als c Daten, dann werden die folgenden Schritte ausgeführt:

Schritt 1: Aus M wird ein proportionaler Anteil der Daten eliminiert. Es ergibt sich eine Menge M' der verbleibenden Daten.

Schritt 2: P(M') wird rekursiv errechnet.

Schritt 3: P(M) wird aus P(M') abgeleitet.

Voraussetzung für die Anwendung des Eliminations-Prinzips ist die Existenz von M', sodaß P(M) ohne großen Aufwand aus P(M') berechnet werden kann. Diese Voraussetzung ist bei der Berechnung einer speziellen Brücke in Kapitel 4.3 gegeben. Die klassische Anwendung dieses Prinzips ist das Bestimmen der mittleren Zahl einer gegebenen unsortierten Folge, siehe z.B. [Aho 74].

Der schwierigste Teil von Algorithmen, die nach dem Prinzip des Eliminierens arbeiten, ist die Lösung von Schritt 1, d.h. das Bestimmen jener Daten, die nicht weiter wichtig sind.

5.3 Plane-sweep Technik

Dieses Prinzip führt bei vielen geometrischen Problemen für ebene Objekte zu effizienten Verfahren. Intuitiv wird die Ebene mit einer vertikalen Geraden V von links nach rechts durchwandert. Mit V wird eine Datenstruktur D mitgeführt, die während der Wanderung manipuliert wird. Manipulationen sind immer notwendig, wenn V auf ein neues Objekt stößt, oder wenn V ein Objekt verläßt. Sei $M = \{O_1, \ldots O_n\}$ eine Menge von n Objekten in der Ebene und sei x_i^l, bzw. x_i^r, die x-Koordinate eines äußerst linken, bzw. rechten, Punktes von O_i, für $1 \leq i \leq n$. Die Grobstruktur jedes Algorithmus, der nach der plane-sweep Technik arbeitet, kann wie folgt dargestellt werden:

Schritt 1: Die Werte x_i^l und x_i^r, für $1 \leq i \leq n$, werden sortiert. Sei $x_1, x_2, \ldots x_{2n}$ die resultierende sortierte Folge F. Jeder Wert in F steht weiterhin mit dem Objekt, von dem es stammt, in Verbindung.

Schritt 2: F wird sukzessiv von links nach rechts abgearbeitet und für jeden x-Wert wird die Datenstruktur D entsprechend manipuliert.

Diese Technik erweist sich dann als nützlich, wenn die Operationen pro x-Wert rasch ausgeführt werden können. Z.B sind

"Füge O_i in D ein, wenn x_i^l verarbeitet wird"

"Entferne O_i aus D, wenn x_i^r verarbeitet wird"

typische Operationen bei Anwendungen der plane-sweep Technik. Sie können z.B. in Kapitel 3.1 gefunden werden, wo die Technik für einen Algorithmus verwendet wird, der die sich schneidenden Paare einer Menge von achsenparallelen Rechtecken in der Ebene bestimmt.

5.4 Geometrische Transformationen

Dieses Hilfsmittel für den Entwurf von Algorithmen unterscheidet sich grundlegend von den oben erklärten Prinzipien. Die Anwendung von geometrischen Transformationen, die die Daten verändern oder vielleicht nur neu interpretieren, führt zu neuen Einsichten und Möglichkeiten, die unter Umständen effizientere Lösungen zulassen. Beispiele dafür sind in Kapitel 4.1 und 4.2 beschrieben worden.

In Kapitel 4.1 werden Punkte zu Geraden und Geraden zu Punkten transformiert. Diese Transformation, die zu dualen Situationen im ursprünglichen Raum und im transformierten Raum führt, erwies sich bereits für viele geometrische Probleme als nützlich.

Ebenso führte die Tranformation, die in Kapitel 4.2 kurz dargestellt wurde, bei mehreren Problemen zu effizienten Lösungen. Die Grundelemente dieser Tranformation sind die Einbettung von ebenen Objekten in drei Dimensionen und die anschließende Invertierung der erhaltenen dreidimensionalen Objekte. Wir verweisen auf [Brown 80], der weitere Beispiele

von Anwendungen geometrischer Transformationen für effiziente algorithmische Lösungen beschreibt.

5.5 Locus-approach

Die Idee des Locus-approach ist weniger als Prinzip für das Entwerfen von Algorithmen und Datenstrukturen zu verstehen. Vielmehr verbirgt sich hinter dieser Bezeichnung eine allgemeine Idee, die sich oft als hilfreich erweist. Es geht dabei um die Berechnung des geometrischen Ortes von identischen Antworten oder Ergebnissen. Z.B. führte diese Idee zum Voronoi Diagramm als Hilfsstruktur für das Postamt Problem, siehe Kapitel 3.2. Jedem Punkt p der gegebenen Menge wird der geometrische Ort jener Punkte zugeordnet, deren nächster Punkt p ist. In Kapitel 4.2 wird jedem gewichteten Punkt p der Ort jener Punkte zugeordnet, deren gewichtet nächster Punkt p ist. In Kapitel 4.1 wird die Ebene durch einen Polygonzug in zwei Teile getrennt, sodaß die Antwort davon abhängt, in welchen Teil ein Suchpunkt fällt.

5.6 Dynamisierungsmethoden

In Kapitel 3.4 wurden zwei voneinander sich grundsätzlich unterscheidende Dynamisierungsmethoden durch Beispiele belegt. Die erste Methode bedient sich balanzierter Bäume und die spezielle Lösung hängt sehr stark von gewissen Eigenschaften des Problems und der verwendeten Bäume ab. Die zweite Methode läßt sich ohne wesentliche Änderungen zur Dynamisierung verschiedenster Datenstrukturen verwenden. Aus diesem Grund gehen wir etwas näher auf sie ein.

Sei D eine statische Datenstruktur, die ein Suchproblem P löst, und sei M die Menge der Daten, die in D gespeichert sind. Um P dynamisch zu lösen, d.h. es können ohne großen Aufwand neue Daten in M eingefügt oder Daten aus M entfernt werden, wird M nicht in einer einzigen Datenstruktur, sondern

in einem System von Datenstrukturen gespeichert. Dazu wird M in disjunkte Teilemengen M_1, ... M_k zerlegt. Jede dieser Teilmengen M_i ist in einer Datenstruktur D_i nach Vorbild von D gespeichert. Der Vorteil des Systems von Datenstrukturen ist die Möglichkeit, eine Einfügung oder eine Entfernung durch Neuaufbau einer einzigen Datenstruktur im System zu reflektieren. Um eine Anfrage für ein Suchobjekt q zu beantworten, muß allerdings jede der Datenstrukturen im System untersucht werden, und die verschiedenen Antworten müssen zu einer einzigen kombiniert werden. Diese letzte Anforderung führt zu einer Charakterisierung all jener Suchprobleme P, die mittels Systemen von Datenstrukturen dynamisiert werden können:

> Sei M die Datenmenge und M_1 und M_2 eine beliebige disjunkte Zerlegung von M, dann muß die Antwort A(M,q) für ein Suchobjekt q in konstanter Zeit von $A(M_1,q)$ und $A(M_2,q)$ abgeleitet werden können.

Aus Platzgründen verzichten wir auf die Behandlung der wichtigen Fragen, wie die Aufteilung der Datenmenge zu wählen ist, und wie das System aufrechterhalten werden kann. Einige Antworten auf diese Fragen werden in [Bentley 79], [Maurer 79b], [Van Leeuwen 80] und [Overmars 81] angeboten.

Literaturhinweise

[Adelson-Velskii 62] Adelson-Velskii,G.M. und Landis,E.M.: Ein Algorithmus zur Informationsorganisation (russisch). Doklady Akad. Nauk SSR 146, 263-266.

[Aho 74] Aho,A.V., Hopcroft,J.E. und Ullman,J.D.: The design and analysis of computer algorithms. Addison-Wesley, Reading.

[Aurenhammer 83] Aurenhammer,F. und Edelsbrunner,H.: An optimal algorithm for constructing the weighted Voronoi diagram in the plane. Report 109, Inst. für Inform., Techn. Univ. Graz, Österreich.

[Bentley 75] Bentley,J.L.: Multidimensional binary search trees used for associative searching. Comm. of the ACM 18, 509-517.

[Bentley 79] Bentley,J.L.: Decomposable searching problems. Inf. Proc. Lett. 8, 244-251.

[Bentley 80a] Bentley,J.L. und Maurer,H.A.: Efficient worst-case data structures for range searching. Acta Inf. 13, 155-168.

[Bentley 80b] Bentley,J.L.: Multidimensional divide-and-conquer. Comm. of the ACM 23, 214-229.

[Bentley 80c] Bentley,J.L. und Wood,D.: An optimal worst case algorithm for reporting intersections on rectangles. IEEE Trans. on Comp. C-39, 571-577.

[Brillinger 72] Brillinger,P.C. und Cohen,D.J.: Introduction to data structures and non-numeric computation. Prentice Hall, Englewood Cliffs.

[Brown 80] Brown,K.Q.: Geometric transforms for fast geometric algorithms. Report CMU-CS-80-101, Dep. of Comp. Sci., Carnegie-Mellon Univ., Penn.

[Coleman 78] Coleman,D.: A structured programming approach to data. MacMillan Press, London.

[Dobkin 76] Dobkin,D.P. und Lipton,R.J.: Multidimensional searching problems. SIAM J. on Comp. 5, 181-186.

[Dyer 82] Dyer,M.E.: Two-variable linear programs are solvable in linear time. Report, Dep. of Math. and Stat., Teesside Pol., UK.

[Edelsbrunner 80a] Edelsbrunner,H.: Dynamic rectangle intersection searching. Report F47, Inst. für Inform., Techn. Univ. Graz, Österreich.

[Edelsbrunner 80b] Edelsbrunner,H.: A time- and space-optimal solution for the planar all intersecting rectangles problem. Report F50, Inst. für Inform., Techn. Univ. Graz, Österreich.

[Edelsbrunner 81] Edelsbrunner,H.: Reporting intersections of geometric objects by means of covering rectangles. EATCS Bulletin 13, 7-11.

[Edelsbrunner 82a] Edelsbrunner,H. und Welzl,E.: On the number of line-separations of a finite set in the plane. Report F97, Inst. für Inform., Techn. Univ. Graz, Österreich.

[Edelsbrunner 82b] Edelsbrunner,H. und Welzl,E.: Halfplanar range estimation. Report F98, Inst. für Inform., Techn. Univ. Graz, Österreich.

[Fredman 80] Fredman,M.L.: The inherent complexity of dynamic data structures which accommodate range queries. Proc. 21st Ann. IEEE Symp. on Found. of Comp. Sci., 191-199.

[Graham 72] Graham,R.L.: An efficient algorithm for determining the convex hull of a finite planar set. Inf. Proc. Lett. 1, 132-133.

[Grünbaum 67] Grünbaum,B.: Convex polytopes. Interscience, London.

[Grünbaum 72] Grünbaum,B.: Arrangements and spreads. Amer. Math. Soc., Providence.

[Horowitz 78] Horowitz,E. und Sahni,S.: Fundamentals of computer algorithms. Computer Science Press, Potomac.

[Jarvis 73] Jarvis,R.A.: On the identification of the convex hull of a finite set of points in the plane. Inf. Proc. Lett. 2, 18-21.

[Jensen 75] Jensen,K. und Wirth,N.: PASCAL - user manual and report. Springer, New York.

[Kirkpatrick 81] Kirkpatrick,D.G.: Optimal search in planar subdivisions. Report 81-13, Dep. of Comp. Sci., Univ. of British Columbia.

[Kirkpatrick 82] Kirkpatrick,D.G. und Seidel,R.: The ultimate planar convex hull algorithm? Proc. 20th Ann. Allerton Conf. on Comm., Contr., and Comp.

[Knuth 68] Knuth,D.E.: Fundamental algorithms. The art of computer programming I. Addison-Wesley, Reading.

[Knuth 69] Knuth,D.E.: Seminumerical algorithms. The art of computer programming II. Addison-Wesley, Reading.

[Knuth 73] Knuth,D.E.: Sorting and searching. The art of computer programming III. Addison-Wesley, Reading.

[Lee 77a] Lee,D.T. und Wong,C.K.: Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. Acta Inf. 9, 23-29.

[Lee 77b] Lee,D.T. und Preparata,F.P.: Location of a point in a planar subdivision and its applications. SIAM J. on Comp. 6, 594-606.

[Maurer 74] Maurer,H.A.: Datenstrukturen und Programmierverfahren. Teubner, Stuttgart.

[Maurer 76] Maurer,H.A. und Wood,D.: Zur Manipulation von Datenmengen. Angew. Inf. 4, 143-149.

[Maurer 79a] Maurer,H.A. und Ottmann,Th.: Manipulating sets of points - a survey. In: Nagl,M. und Schneider,H.-J. (Ed.): Applied computer science 13, 9-29, Carl Hanser.

[Maurer 79b] Maurer,H.A. und Ottmann,Th.: Dynamic solutions of decomposable searching problems. In: Pape,U. (Ed.): Discrete structures and algorithms, 17-24, Carl Hanser.

[McCreight 80] McCreight,E.M.: Efficient algorithms for enumerating intersecting intervals and rectangles. Report CSL-80-9, XEROX Parc, California.

[Nievergelt 73] Nievergelt,J. und Reingold,E.M.: Binary search trees of bounded balance. SIAM J. on Comp. 2, 33-43.

[Overmars 81] Overmars,M.H. und van Leeuwen,J.: Worst-case optimal insertion and deletion methods for decomposable searching problems. Inf. Proc. Lett. 12, 168-173.

[Pfaltz 77] Pfaltz,J.L.: Computer data structures. McGraw-Hill, New York.

[Preparata 77] Preparata,F.P. und Hong,S.J.: Convex hulls of finite sets of points in two and three dimensions. Comm. of the ACM 20, 87-93.

[Preparata 81] Preparata,F.P.: A new approach to planar point location. SIAM J. on Comp. 10, 473-482.

[Prim 57] Prim,R.C.: Shortest connection networks and some generalizations. Bell Sys. Tech. J. 36, 1389-1401.

[Seidel 81] Seidel,R.: A new convex hull algorithm optimal for point sets in even dimensions. Report 81-14, Dep. of Comp. Sci., Univ. of British Columbia.

[Shamos 75a] Shamos,M.I.: Geometric complexity. Proc. 7th Ann. ACM Symp. on Th. of Comp., 224-233.

[Shamos 75b] Shamos,M.I. und Hoey,D.: Closest-point problems. Proc. 16th Ann. IEEE Symp. on Found. of Comp. Sci., 151-162.

[van Leeuwen 80] van Leeuwen,J. und Wood,D.: Dynamization of decomposable searching problems. Inf. Proc. Lett. 10, 51-56.

[Wettstein 72] Wettstein,H.: Systemprogrammierung. Carl Hanser, München.

[Willard 78] Willard,D.E.: Predicate-oriented database search algorithms. Report TR-20-78, Aiken Comp. Lab., Harvard Univ., Mass.

[Willard 82] Willard,D.E.: New data structures for orthogonal queries. To appear in SIAM J. on Comp.

[Wirth 76] Wirth,N.: Algorithms + data structures = programs. Prentice Hall Englewood Cliffs.