4. Effiziente Algorithmen + Datenstrukturen

→ siehe gleichnamiges Buch von N. Wirth

Für gegebene Aufgabenstellungen ist es wichtig, geschickt Datenstrukturen zu wählen und Algorithmen, die darauf effizient arbeiten.

Wichtig bei der Wahl einer Datenstruktur ist dabei die Frage, welche Zugriffsoperation möglichst effizient ausgeführt werden soll.

4.1. Diskussion ausgewählter Algorithmen

Sortieren ist eine der wichtigsten Aufgaben in der betriebswirtschaftlichen Informatik.

4.1.1. Komplexität von Sortierverfahren

Gängige Sortierverfahren:

Aufwand		
n^2	Insertsort	Sortieren durch Einsortieren
n^2	Selectsort	—— " —— Auswählen
$n \log n$	Mergesort	—— " —— Mischen
$n \log n$	Quicksort	—— " —— Zerteilen
$n \log n$	Heapsort	—— " —— Auswahlbäume
n^2	Bubblesort	—— " —— Vertauschen

Beispiel: Bubblesort
Gegeben Feld var [1:n] array int a
Folgendes Programm sortiert die Elemente im Feld a aufsteigend:

```
var nat i := 1;
while i < n
do if a[i] > a[i+1]
   then a[i], a[i+1] := a[i+1], a[i];
        if i > 1 then i := i-1
                 else i := i+1 (i := 2)
        fi
   else i := i+1
   fi
od
```

Leichte Elemente steigen wie Bubbles auf.

Wir stellen uns nun die Frage, mit welchem Aufwand wir eine Sequenz sortieren können.

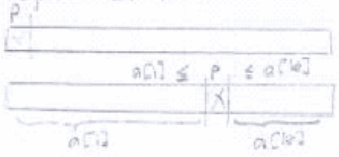
Bubblesort hat im Allgemeinen den Aufwand von n^2 Schritten, ist also von $O(n^2)$.

Wichtige Unterschiede in der Aufwandsabschätzung
- Aufwand im ungünstigsten Fall (worst case)
- durchschnittlicher Aufwand (bei gegebener Wahrscheinlichkeitsverteilung auf den Eingaben)

Mergesort: Aufwand jedes Mal halbiert: $n \log n$
Heapsort: Aufwand beim aufstellen und elemente rausholen: $n \log n$

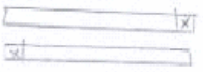
Aufwand bei Quicksort

Prinzip von Quicksort



p: Vergleichselement (= Pivotelement) wählen

worst case:



d.h. es sind immer alle Elemente kleiner oder alle Elemente größer als das Pivot-Element.

Im besten Fall läge das Pivot-Element so, dass die Hälfte der Elemente größer und die Hälfte der Elemente kleiner ist. Im Durchschnitt kann mit einer $\frac{n}{2} - \frac{n}{3}$ Aufteilung gerechnet werden.

→ Aufwand im worst case: n^2
Durchschnitt: $n \log n$

Welches Verfahren gewählt wird, hängt von best. Rahmenbedingungen a
- Größe der Sequenz:
  Sind die Sequenzen sehr klein, so empfehlen sich einfache, speziell darauf zugeschnittene Verfahren.
- Kenntnisse über Anordnung der Elemente (Wahrscheinlichkeitsverteilung der Eingabe): Sind die Elemente in der zu sortierenden Sequenz nicht zufällig angeordnet sondern beispielsweise schon weitgehend vorsortiert, so arbeiten bestimmte Verfahren gut, andere extrem schlecht.
- Nebenbedingung techn. Art: Wie sind die Sequenzen gespeichert. Wird im Hauptspeicher oder im Hintergrundspeicher sortiert.
  interner Sortieren — externes Sortieren

4.12. Wege in Graphen

Viele praktische Fragestellungen führen auf Probleme, die sich mit Wegen in Graphen beschreiben lassen. Wir behandeln stellvertretend einen Algorithmus zum Finden von Wegen in einem gerichteten Graphen. "Warshalls Algorithmus"

Gegeben Knotenmenge $V = \{1, \dots, n\}$
Kanten $c: V \times V \to \mathbb{B}$

$c(i,j)$ ~ es existiert eine Kante von i nach j
(Adjazenzmatrix c)

Wir rechnen die transitive Hülle aus

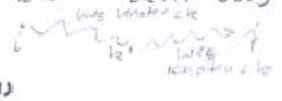
```
fct th = (nat i, nat j) bool:
  "es ex. ein Weg in c von i nach j"
      wth (i,j,n)
```

Idee: wth (i,j,k) bestimmt die Antwort auf die Frage
"Ex. ein Weg in c von i nach j über innere Knoten ≤ k"

Alg. Warshall Transitive Hülle

```
fct wth = (nat i,j,k) bool:
    if k = 0 then c(i,j)
             else wth(i,j,k-1)
                  v (wth(i,k,k-1) ∧
                     wth(k,j,k-1))
    fi
```

Fallunterscheidung:
Für wth(i,j,k) mit k>0
Fallunterscheidung: jeder Weg mit Knoten ≤ k erfüllt einen der folgenden Fälle
(1) k ist nicht in dem Weg; dann gilt wth(i,j,k-1)
(2) k ist in dem Weg



ese Idee können wir in einen Algorithmus nwandeln, der auf einem zweidimensionalen Feld a arbeitet:

```
{∀ i,j : 1 ≤ i,j ≤ n ⇒ a[i,j] = c(i,j)}
for k := 1 to n do
  for i := 1 to n do
    for j := 1 to n do
      a[i,j] := a[i,j] v (a[i,k] ∧ a[k,j])
    od
  od
od
{∀ i,j : 1 ≤ i,j ≤ n : a[i,j] = wth(i,j,n)}   = th(i,j)
```

vgl. oben

omplexität n^3

t.2. Bäume

Baumstrukturen sind ein Kernthema der Informatik.

t.2.1. Geordnete, orientierte, sortierte Bäume

Es existieren zwei Blickweisen auf Bäume:

(1) Bäume als Datenstruktur mit rekursivem Aufbau und Zugriffsfunktionen

(2) Bäume als Graphen.

Zu Bäumen als Graphen:
Ein ungerichteter Graph G heißt (nichtorientierter) Baum, falls G
- zyklenfrei
- zusammenhangend

ist.

d.h. keine Richtungen

ungerichteter Graph: Zyklen vorhanden und muss Ewig. sein

n gerichteter Graph heißt orientierter Baum, falls

- eine Wurzel k ex. (ein Knoten k)
- von k aus sind alle anderen Knoten erreichbar, und zwar auf genau einem Weg (d.h. jeder Knoten hat höchstens eine eingehende Kante)

Knoten eine Reihenfolge für seine Teilbäume gegeben ist.

sort ordtree = ordtree (m root, seq ordtree subtrees)

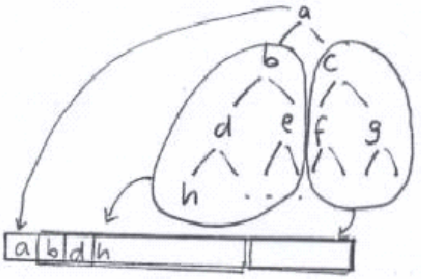
Verzweigungsgrad = max. Anzahl der Teilbäume pro Knoten

Vollständiger Baum: Alle Wege (zu Blättern von der Wurzel) haben gleiche länge.

4.2.2. Darstellung von Bäumen durch Felder

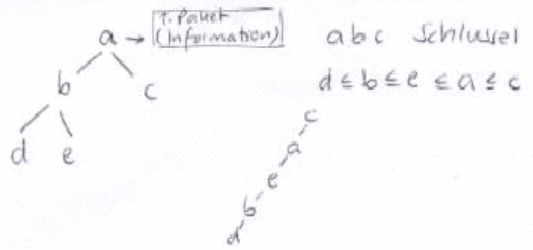
Üblicherweise werden Bäume durch verkettete Zeigerstrukturen in Rechnern dargestellt

Wir können vollständige Bäume auch „statisch" in Feldern ablegen (siehe Abb.



4.2.3. AVL-Bäume

Beim Arbeiten mit Bäumen sind wir an kurzem (effizienten) Zugriff auf die Elemente interessiert.



Dies ist nur gewährleistet, wenn die Bäume nicht entarten, d.h. wenn die Höhe des Baumes logarithmisch mit der Anzahl der Knoten im Baum wächst.
Dies könnte auf die Forderung hinauslaufen, dass alle Wege von der Wurzel zu den Blättern sich höchsten um 1 in der Länge unterscheiden

Allerdings wird es dann sehr aufwendig beim Einfügen oder Löschen von Knoten die Ausgewogenheit zu erhalten.

Ausweg:

Statt völliger Ausgewogenheit fordern wir, dass sich die Höhen der linken und rechten Teilbaums maximal um 1 unterscheiden.
↪ das liefert dann die AVL-Bäume)

ef. AVL: sortierter Binär-Baum, bei dem sich in allen Knoten die Höhen der Teilbäume nur um max. 1 unterscheiden.

Sortiert: Für alle Teilbäume gilt: Die Einträge im linken Teilbaum sind kleiner gleich (≤) die Wurzel und die Wurzel ist kleiner gleich (≤) den Einträgen in den rechten Teilbäumen.
⇒ Sequenz Inordnung ist sortiert.

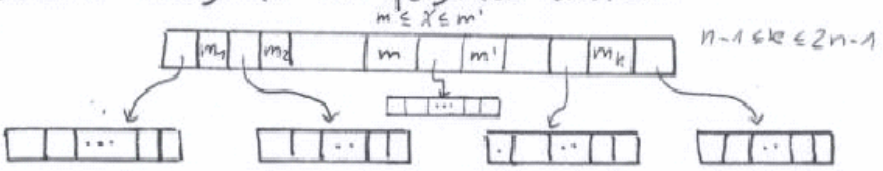
Vorteil der Sortierung: gesteuerte Suche möglich.

Wichtig: Bei AVL-Bäumen können Knoten mit geringem Aufwand eingefügt oder gelöscht werden (siehe später).

4.2.4. B-Bäume

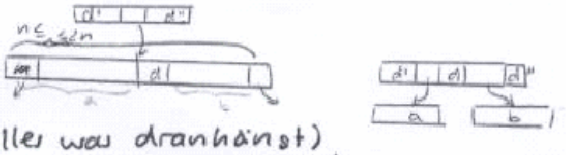
B-Bäume sind die Grundlage für die Speicherung von Informationen in Datenbanken.
1970 von Prof. R. Bayer (TUM) vorgeschlagen!

Sei $n \in \mathbb{N}$ eine gegebene Zahl. Ein B-Baum über einer linear geordneten Menge M hat folgende Gestalt:



Ein B-Baum ist leer, oder enthält zwischen n und 2n Teilbäume. Ein B-Baum ist sortiert.

Vorteile: (1) Es kann gezielt gesucht werden
(2) Die Höhe (Anzahl der Suchschritte) ist extrem klein. (Beim Speichern des B-Baums auf Hintergrundmedien sind nur wenige Seitenzugriffe erforderlich)
(3) Einfügen (oder Löschen) neuer Elemente extrem billig.



Bemerkung:
Die obere „Bürste" (alles was dranhängt) kann weniger als n Einträge enthalten.

In B-Bäumen haben alle Wege von der Wurzel zu den Blättern die gleiche Länge

---

4.3. Effiziente Speicherung großer Datenmengen

{a,b,c} {b,c,a} {b,b,c,a}

Die Datenstruktur der (endlichen) Mengen wird in den meisten Programmiersprachen nicht direkt unterstützt. Im Folgenden betrachten wir Datenstrukturen für die effiziente Behandlung großer Mengen.

4.3.1. Rechenstruktur der Mengen (mit Zugriff über Schlüssel)

Wir nehmen an, dass wir Datensätze der Sorte *data* speichern wollen. Für jeden Datensatz setzen wir einen Schlüssel voraus:

fct key = (data) key MTNo (238413, Otto, Huber, 100371 (Hrbr), ...)

Wir wollen eine endl. Menge von Daten speichern und über Schlüssel darauf zugreifen.
Die Sorte *store* bezeichnet die Menge der endl. Mengen von Daten. Zunächst interessiert uns die Struktur der Elemente der Sorte *store* nicht, sondern die Operationen, die wir zur Verfügung haben.

fct emptystore = store "erzeugt leeren store"
fct get = (store, key) data
fct insert = (store, data) store
fct delete = (store, key) store

Logische Festlegungen der Wirkungsweise dieser Funktionen (Schnittstellenverhalten):

k = key(d) ⇒ get(insert(s,d), k) = d
k ≠ key(d) ⇒ get(insert(s,d), k) = get(s,k)
delete(emptytree, k) = emptytree
k = key(d) ⇒ delete(insert(s,d), k) = ~~s oder~~ delete(s,k)
k ≠ key(d) ⇒ delete(insert(s,d),k) = insert(delete(s,k), d)
get(~~empty~~tree, k) = nicht spezifiziert

wir nehmen delete(s,k) an (*)

Zusätzliche Operationen

fct isempty = (store) bool
fct isempty = (store, key) bool
isempty(emptytree, k) = false
isempty(insert(s,d), k) = (k = key(d) ∨ isempty(s,k))

fast ausgegl. Baum

AVL-Baum

43.2. Mengendarstellung durch AVL Bäume

Wir stellen nun die Elemente der Sorte store durch AVL-Bäume dar.

sort store = tree data

Binärbäume

Damit sind bestimmte Funktionen wie root, left, right auf store vorgegeben. Die weiteren (eigentlichen) Funktionen auf store programmieren wir nun, abgestützt auf diesen Grundfunktionen.

```
fct get = (store s, key k) data:
    if  key(root(s)) = k  then root(s)                    // Datensatz hat den richtigen Schlüssel
    elif k < key(root(s)) then get(left(s), k)            // Suche im linken Teilbaum
                          else get(right(s), k)           //   "   "  rechten   "
    fi
```

Baumkonstruktor

```
fct insert = (store s, data d) store:
    if  s = emptytree then cons(emptytree, d, emptytree)   hier kommt uns die
    elif key(d) = key(root(s)) then cons(left(s), d, right(s))   (*) Voraussetzung zugute
    elif key(d) < key(root(s)) then cons(insert(left(s), d), root(s), ~~right~~ right(s))
                               else cons(left(s), root(s), insert(right(s), d))
    fi
```

Achtung: insert liefert sortierten Baum, aber nicht notwendigerweise balancierten Baum, selbst wenn s balanciert ist.

⇒ Abb. S. 343!