Effiziente Algorithmen

Martin Hofmann und Jan Johannsen

Institut für Informatik

LMU München

Sommersemester 2002

Organisatorisches

- Vorlesung: Mi 10 - 12 Uhr, HS 122 ; Fr 10 - 12 Uhr, HS 201
- Übungen: Mo 14-18, Di 14-20, Do 14-18. Bitte in Listen eintragen. Beginn 22.4.
- Klausur: 20.7. 9-12. Voraussetzung $\geq 50\%$ bei Hausaufgaben.
- Buch zur Vorlesung: T. H. Cormen, C. E Leiserson and R. L. Rivest, Introduction to Algorithms, MIT Press (1990).

Literatur

- A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.*
- T. Cormen, C. Leiserson and R. Rivest, Introduction to Algorithms, MIT Press, 1990.
- D. Knuth. *The Art of Computer Programming*, vol I-III, Addison-Wesley, 1968.
- H. Kriegel. *Vorlesungsskript zu „Effiziente Algorithmen“*. Im Sekretariat DBS erhältlich.
- U. Schöning. *Algorithmik*, Spektrum Verlag, 2001.
- R. Sedgewick. *Algorithms.* Addison-Wesley, 1988.

Was sind Algorithmen, wieso Algorithmen?

Ein Algorithmus ist eine genau festgelegte Berechnungsvorschrift, die zu einer Eingabe oder Problemstellung eine wohldefinierte Ausgabe oder Antwort berechnet.

Ein Algorithmus kann durch ein Programm in einer Programmiersprache angegeben werden, aber typischerweise lässt sich ein und derselbe Algorithmus auf verschiedene Arten und in verschiedenen Programmiersprachen implementieren.

Neben Programmiermethodiken (wie Modularisierung, Objektorientierung, etc.) bilden Algorithmen den Schlüssel zu erfolgreicher Softwareentwicklung. Durch Verwendung professioneller Algorithmen lassen sich oft spektakuläre Laufzeitgewinne realisieren.

Zwar stellen Sprachen wie Java viele Algorithmen fertig implementiert in Form von Bibliotheken bereit; viele Projekte werden jedoch nach wie vor in C implementiert, außerdem ist es oft der Fall, dass eine Bibliotheksfunktion nicht genau passt.

Die Algorithmik ist eine lebendige Disziplin. Neue Problemstellungen (Bioinformatik, Internet, Mobilität) erfordern zum Teil neuartige Algorithmen. Für alte Problemstellungen werden bessere Algorithmen entwickelt (kombinatorische Optimierung, Planung). Schnellere Hardware rückt bisher unlösbare Probleme in den Bereich des Machbaren (automatische Programmverifikation, Simulation).

Zusammenfassung

- Einführung: Sortieren durch Einfügen, Sortieren durch Mischen, Laufzeitabschätzungen, Lösen von Rekurrenzen.
- Sortieren und Suchen: Heapsort, Quicksort, Maximum und Minimum, Ordnungsstatistiken, Median.
- Professionelle Datenstrukturen: Balancierte Binärbäume, Hashtabellen, *union find*, *priority queues*.
- Entwurfsmethoden: Divide-and-conquer, Greedy Algorithmen, dynamische Programmierung, amortisierte Analyse.
- Graphalgorithmen: transitive Hülle, Spannbäume, kürzester Pfad, Fluss in Netzwerken,
- Schnelle Fouriertransformation: Komplexe Zahlen, Rechnen mit Polynomen, FFT.
- Geometrische Algorithmen: Schnittpunktbestimmung, konvexe Hülle, Entwurfsmethode *sweep line*.
- Zeichenketten: Finden von Substrings, Editierdistanz.
- Randomisierung: Primzahltests, Monte-Carlo-Methode.

Sortieren

Eingabe: Eine Folge von n Zahlen $\langle a_1, a_2, \ldots, a_n \rangle$.

Ausgabe: Eine Permutation (Umordnung) $\langle a_{\pi 1}, a_{\pi 2}, \ldots, a_{\pi n} \rangle$, sodass $a_{\pi 1} \leq a_{\pi 2} \leq \cdots \leq a_{\pi n}$.

Beispiel: $\langle 31, 41, 59, 26, 41, 58 \rangle \mapsto \langle 26, 31, 41, 41, 58, 59 \rangle$.

Allgemeiner: „Zahlen“ $\mapsto$ „Objekte“, „$\leq$“ $\mapsto$ „Ordnungsrelation“.

Beispiel: Zeichenketten, lexikographische Ordnung.

Sortieren durch Einfügen

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1..j − 1]
4          i ← j − 1
5          while i > 0 and A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← key
```

Pseudocode

Um von Implementierungsdetails zu abstrahieren, verwenden wir zur Angabe von Algorithmen Pseudocode: Eine PASCAL oder C ähnliche Sprache, welche

- Neben formellen Anweisungen und Kontrollstrukturen auch Umgangssprache enthalten kann,
- Verallgemeinerte Datentypen (wie Mengen, Graphen, etc.) bereitstellt,
- Blockstruktur auch durch Einrückung kennzeichnet (wie in Python)

Laufzeitanalyse

Wir wollen die Laufzeit eines Algorithmus als Funktion der Eingabegröße ausdrücken.

Manchmal auch den Verbrauch an anderen Ressourcen wie Speicherplatz, Bandbreite, Prozessorzahl.

Laufzeit bezieht sich auf ein bestimmtes Maschinenmodell, hier RAM (*random access machine*).

Laufzeit kann neben der Größe der Eingabe auch von deren Art abhängen (*worst case*, *best case*, *average case*). Meistens *worst case*.

Laufzeit von Insertion-Sort

Insertion-Sort(A)	Zeit	Wie oft?
1 **for** $j \leftarrow 2$ **to** $length[A]$	c_1	n
2 **do** $key \leftarrow A[j]$	c_2	$n-1$
3 ▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$		
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] \leftarrow key$	c_8	$n-1$

t_j = Anzahl der Durchläufe der *while*-Schleife.

$c_1 - c_8$ = unspezifizierte Konstanten.

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)$$

Bester Fall: Array bereits sortiert

Ist das Array bereits aufsteigend sortiert, so wird die *while*-Schleife jeweils nur einmal durchlaufen: $t_j = 1$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)$$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Also ist $T(n)$ eine lineare Funktion der Eingabegröße n.

Schlechtester Fall: Array absteigend sortiert

Ist das Array bereits absteigend sortiert, so wird die *while*-Schleife maximal oft durchlaufen: $t_j = j$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)$$

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \qquad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1\right) + (c_6 + c_7)\frac{n(n-1)}{2}$$

Also ist $T(n)$ eine quadratische Funktion der Eingabegröße n.

worst case und *average case*

Meistens geht man bei der Analyse von Algorithmen vom schlechtesten Fall (*worst case*) aus.

- *worst case* Analyse liefert obere Schranken
- In vielen Fällen ist der *worst case* die Regel
- Der aussagekräftigere (gewichtete) Mittelwert der Laufzeit über alle Eingaben einer festen Länge (*average case*) ist oft bis auf eine multiplikative Konstante nicht besser als der *worst case*.
- Manchmal muss man aber eine *average case* Analyse durchführen (Beispiel: Quicksort)
- Manchmal zeigt sich, dass in der Praxis der *worst case* selten auftritt (Beispiel: Simplexverfahren, Typinferenz in ML)

Größenordnungen

Um Rechnungen zu vereinfachen und da Konstanten wie $c_1, \ldots, c_8$ sowieso willkürlich sind, beschränkt man sich oft darauf die Größenordnung der Laufzeit anzugeben:

$$an^2 + bn + c = \Theta(n^2)$$

$$an + b = \Theta(n)$$

$$a2^n + bn^{10000} = \Theta(2^n)$$

$\Theta(f(n))$ bezeichnet alle Funktionen der Größenordnung $f(n)$. Dies wird demnächst formal definiert.

Laufzeit von Insertion-Sort im schlechtesten Fall ist $\Theta(n^2)$.

Teile und herrsche

Das Entwurfsverfahren *divide-and-conquer* (Teile und Herrsche, *divide et impera*) dient dem Entwurf rekursiver Algorithmen.

Die Idee ist es, ein Problem der Größe n in mehrere gleichartige aber kleinere Teilprobleme zu zerlegen (*divide*).

Aus rekursiv gewonnenen Lösungen der Teilprobleme gilt es dann, eine Gesamtlösung für das ursprüngliche Problem zusammenzusetzen (*conquer*).

Beispiel: Sortieren durch Mischen (*merge sort*):

Teile n-elementige Folge in zwei Teilfolgen der Größe $n/2 \pm 1$.

Sortiere die beiden Teilfolgen rekursiv.

Füge die nunmehr sortierten Teilfolgen zusammen durch Reißverschlussverfahren.

Sortieren durch Mischen

```
MERGE-SORT(A, p, r)
    ▷ Sortiere A[p..r]
1   if p < r
2       then q ← ⌊(p + r)/2⌋
3       MERGE-SORT(A, p, q)
4       MERGE-SORT(A, q + 1, r)
5       MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
    ▷ Sortiere A[p..r] unter der Annahme, dass A[p..q] und A[q + 1..r] sortiert sind.
1   i ← p; j ← q + 1
2   for k ← 1 to r − p + 1
3       do if A[i] ≤ A[j] then B[k] ← A[i]; i ← i + 1 else B[k] ← A[j]; j ← j + 1
4   for k ← 1 to r − p + 1 do A[k + r − 1] ← B[k]
```

Analyse von Merge-Sort

Sei $T(n)$ die Laufzeit von Merge-Sort.

Das Aufteilen braucht $\Theta(1)$ Schritte.

Die rekursiven Aufrufe brauchen $2T(n/2)$ Schritte.

Das Mischen braucht $\Theta(n)$ Schritte.

Also:

$$T(n) = 2T(n/2) + \Theta(n), \text{ wenn } n > 1$$

NB $T(1)$ ist irgendein fester Wert.

Die Lösung dieser Rekurrenz ist $T(n) = \Theta(n \log(n))$.

Für große n ist das besser als $\Theta(n^2)$ trotz des Aufwandes für die Verwaltung der Rekursion.

Motivation der Lösung

Intuitiv: Rekursionstiefe: $\log(n)$, auf dem Tiefenniveau k hat man 2^k Teilprobleme der Größe jeweils $g_k := n/2^k$, jedes verlangt einen Mischaufwand von $\Theta(g_k) = \Theta(n/2^k)$. Macht also $\Theta(n)$ auf jedem Niveau: $\Theta(n \log(n))$ insgesamt.

Durch Formalismus: Sei $T(n)$ für Zweierpotenzen n definiert durch

$$T(n) = \begin{cases} 2, \text{ wenn } n = 1 \\ 2T(n/2) + n, \text{ wenn } n = 2^k, k > 1 \end{cases}$$

Dann gilt $T(2^k) = k2^k$, also $T(n) = n \log_2(n)$ für $n = 2^k$.

Beweis durch Induktion über k (oder n).

Asymptotik: Definition von Θ

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen. (NB Geht auch für andere Definitionsbereiche.)

$\Theta(g) = \{f \mid \text{es ex. } c_1, c_2 > 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ für alle } n \geq n_0\}$

$\Theta(g)$ ist also eine Menge von Funktionen: die Funktionen derselben Größenordnung wie g.

In Ermangelung guter Notation für Funktionen schreibt man z.B. $\Theta(n^2)$ und meint damit $\Theta(g)$ wobei $g(n) = n^2$.

Bei $\Theta(m + n^2)$ wird's schwierig. Da muss aus dem Zusammenhang klar werden, auf welche Variable sich das Θ bezieht.

Man schreibt $f = \Theta(g)$ anstelle von $f \in \Theta(g)$.

Beispiele:

$$4n^2 + 10n + 3 = \Theta(n^2)$$
$$an + b = \Theta(n)$$
$$a2^n + bn^{10000} = \Theta(2^n)$$

Asymptotik: Ausdrücke mit Θ

Kommt $\Theta(g)$ in einem Ausdruck vor, so bezeichnet dieser die Menge aller Funktionen, die man erhält, wenn man die Elemente von $\Theta(g)$ für $\Theta(g)$ einsetzt.

Z.B. ist $n^2 + 100n + 2\log(n) = n^2 + \Theta(n)$ aber $2n^2 \neq n^2 + \Theta(n)$.

Manchmal kommt Θ sowohl links, als auch rechts des Gleichheitszeichens vor. Man meint dann eine Inklusion der entsprechenden Mengen. M.a.W. jede Funktion links kommt auch rechts vor.

Z.B.

$$2n^2 + \Theta(n) = \Theta(n^2)$$

Die Verwendung von „=“ statt „$\in$“ oder „$\subseteq$“ ist zugegebenermaßen etwas unbefriedigend, z.B. nicht symmetrisch, hat sich aber aus praktischen Gründen durchgesetzt.

Asymptotik: O, Ω, o, ω

Seien wieder $f, g : \mathbb{N} \to \mathbb{R}$. (NB Geht auch für andere Definitionsbereiche.)

$$O(g) = \{f \mid \text{es ex. } c > 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass } 0 \leq f(n) \leq cg(n) \text{ für alle } n \geq n_0\}$$
$$\Omega(g) = \{f \mid \text{es ex. } c > 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass } cg(n) \leq f(n) \text{ für alle } n \geq n_0\}$$

Beachte: $\Theta(g) = O(g) \cap \Omega(g)$.

$f = O(g)$ heisst: g ist eine asymptotische obere Schranke für f.

$f = \Omega(g)$ heisst: g ist eine asymptotische untere Schranke für f.

$$o(g) = \{f \mid \text{für alle } c > 0 \text{ gibt es } n \in \mathbb{N} \text{ so dass } 0 \leq f(n) \leq cg(n) \text{ für alle } n \geq n_0\}$$
$$\omega(g) = \{f \mid \text{für alle } c > 0 \text{ gibt es } n \in \mathbb{N} \text{ so dass } cg(n) \leq f(n) \text{ für alle } n \geq n_0\}$$

$f = o(g)$ heisst: g ist eine asymptotische obere Schranke für f und f ist nicht asymptotisch proportional zu g.

$f = \omega(g)$ heisst: g ist eine asymptotische untere Schranke für f und f ist nicht asymptotisch proportional zu g..

Beachte: $f = o(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ und $f = \omega(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Beispiele für Asymptotik

- $n^2 = o(n^3)$
- $10^{10}n^2 = O(10^{-10}n^2)$ aber nicht mit o statt O.
- $f = o(g)$ impliziert $f = O(g)$
- $\Theta(\log_b(n)) = \Theta(\log_c(n))$ daher schreiben wir einfach $\Theta(\log(n))$. Bei log ist die Basis unspezifiziert, CORMEN schreibt $\lg(x)$ für $\log_2(x)$. Allgemein schreibt man $\ln(x)$ für $\log_e(x)$.
- $f = \Theta(g)$ und $g = \Theta(h)$ impliziert $f = \Theta(h)$. Ähnliches gilt für O, Ω, o, ω.
- $\log n = o(n^\epsilon)$ für jedes $\epsilon > 0$.
- $n! = O(n^n)$
- $\log(n!) = \Omega(n \log n)$
- $\log(n!) = \Theta(n \log n)$
- $\Theta(f) + \Theta(g) = \Theta(f + g)$ und umgekehrt und auch für O, Ω, o, ω.

Asymptotik und Induktion

Will man eine asymptotische Beziehung durch Induktion beweisen, so muss man die Konstante c ein für alle Mal wählen und darf sie nicht während des Induktionsschritts abändern:

Sei $S(n) = \sum_{i=1}^{n} i$, also $S(n) = n(n+1)/2$.

Falscher Beweis von $S(n) = O(n)$:

$S(1) = O(1)$.

Sei $S(n) = O(n)$, dann gilt $S(n+1) = S(n) + (n+1) = O(n) + O(n) = O(n)$.

Asymptotik und Induktion

Richtiger Beweis von $S(n) = \Omega(n^2)$:

Wir versuchen die Induktion mit einem noch zu bestimmenden c durchzukriegen und leiten Bedingungen an diese Konstante ab.

$S(1) = 1$, daraus $c \leq 1$.

Sei $S(n) \geq cn^2$. Es ist
$S(n+1) = S(n) + n + 1 \geq cn^2 + n + 1 \geq cn^2 + 2cn + c = c(n+1)^2$ falls $c \leq 1/2$.

Also funktioniert's mit $c \leq 1/2$, insbesondere $c = 1/2$.

Die Induktion fing bei $n = 1$ an, also können wir $n_0 = 1$ nehmen. □

Bemerkung: Ist die beschränkende Funktion stets größer als Null, so kann man immer $n_0 = 1$ wählen und eventuelle Sprünge der zu beschränkenden Funktion durch Vergrößern/Verkleinern von c auffangen. Bei $n = O(n - 10)$ beispielsweise geht es nicht.

Lösen von Rekurrenzen bei *divide and conquer*

Bei der Analyse von *divide-and-conquer* Algorithmen stößt man auf Rekurrenzen der Form:

$$T(n) = aT(n/b) + f(n)$$

Das passiert dann, wenn ein Problem der Größe n in a Teilprobleme der Größe n/b zerlegt wird und der Aufwand für das Aufteilen und Zusammenfassen der Teilresultate Aufwand $f(n)$ erfordert.

Bemerkung: n/b steht hier für $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$.

Mehr noch: $aT(n/b)$ kann man hier sogar als $a_1 T(\lfloor n/b \rfloor) + a_2 T(\lceil n/b \rceil)$ mit $a_1 + a_2 = a$ lesen.

Die *Master-Methode* liefert eine kochrezeptartige Lösung für derartige Rekurrenzen.

Hauptsatz der Master-Methode

Satz: Seien $a \geq 1, b > 1$ Konstanten, $f, T : \mathbb{N} \to \mathbb{R}$ Funktionen und gelte

$$T(n) = aT(n/b) + f(n)$$

Dann erfüllt T die folgenden Größenordnungsbeziehungen:

1. Wenn $f(n) = O(n^{\log_b a - \epsilon})$ für ein $\epsilon > 0$, so gilt $T(n) = \Theta(n^{\log_b a})$.
2. Wenn $f(n) = \Theta(n^{\log_b a})$, so gilt $T(n) = \Theta(n^{\log_b a} \log(n))$.
3. Wenn $f(n) = \Omega(n^{\log_b a + \epsilon})$ für ein $\epsilon > 0$ und außerdem $af(n/b) \leq cf(n)$ für ein $c < 1$ und genügend großes n, so gilt $T(n) = \Theta(f(n))$.

Zum Beweis, siehe Cormen Abschnitt 4.4.

Beispiele für die Master Methode

- Die Laufzeit $T(n)$ von MERGE-SORT genügt der Beziehung:
  $T(n) = 2T(n/2) + \Theta(n)$ somit $T(n) = \Theta(n^{\log_2(2)} \log(n)) = \Theta(n \log(n))$.
- Wenn $T(n) = 2T(n/3) + n$ dann $T(n) = \Theta(n)$.
- Wenn $T(n) = 2T(n/2) + n^2$ dann $T(n) = \Theta(n^2)$.
- Wenn $T(n) = 4T(n/2) + n$ dann $T(n) = \Theta(n^2)$.
- Die Rekurrenz $2T(n/2) + n \log n$ kann man mit der Master-Methode nicht lösen. Die Lösung ist hier $T(n) = \Theta(n \log^2(n))$.

Matrizenmultiplikation

Seien $A = (a_{ik}), B = (b_{ik})$ zwei $n \times n$ Matrizen. Das Produkt $C = AB$ ist definiert durch

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$$

Das macht $\Theta(n^3)$ Additionen und Multiplikationen.

Matrizen kann man auch blockweise multiplizieren:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Matrizenmultiplikation mit *divide-and-conquer*

$T(n) =$ Anzahl der Operationen erforderlich zur Multiplikation zweier $n \times n$ Matrizen

Es ist mit *divide-and-conquer*:

$$T(n) = 8T(n/2) + \Theta(n^2)$$

Es ist $\log_2(8) = 3$ und $\Theta(n^2) = O(n^{3-\epsilon})$, z.B. mit $\epsilon = 1$, also $T(n) = \Theta(n^3)$.

Keine Verbesserung!

Strassens erstaunlicher Algorithmus

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} R & S \\ T & U \end{pmatrix}$$

wobei (V. Strassens geniale Idee):

$$R = P_5 + P_4 - P_2 + P_6 \qquad S = P_1 + P_2$$
$$T = P_3 + P_4 \qquad U = P_5 + P_1 - P_3 - P_7$$
$$P_1 = A(F - H) \qquad P_2 = (A + B)H$$
$$P_3 = (C + D)E \qquad P_4 = D(G - E)$$
$$P_5 = (A + D)(E + H) \qquad P_6 = (B - D)(G + H)$$
$$P_7 = (A - C)(E + F)$$

Nunmehr ist

$$T(n) = 7T(n/2) + \Theta(n^2)$$

Also: $T(n) = \Theta(n^{\log_2(7)}) = O(n^{2,81})$.

Heapsort

Heapsort ist ein Verfahren, welches ein Array der Größe n ohne zusätzlichen Speicherplatz in Zeit $O(n \log n)$ sortiert.

Dies geschieht unter gedanklicher Verwendung einer baumartigen Datenstruktur, dem *heap*.

Aus einem *heap* kann man in logarithmischer Zeit das größte Element entfernen. Sukzessives Entfernen der größten Elemente liefert die gewünschte Sortierung.

Man kann auch neue Elemente in logarithmischer Zeit einfügen, was alternativ eine Vorgehensweise wie bei Insertion-Sort erlaubt.

Heaps

Ein *heap* (dt. „Halde“) ist ein binärer Baum mit den folgenden Eigenschaften

H1 Die Knoten und die Blätter des Baums sind mit Objekten beschriftet (hier Zahlen).

H2 Alle Schichten sind gefüllt bis auf den rechten Teil der Untersten. M.a.W. alle Pfade haben die Länge d oder $d-1$, hat ein Pfad die Länge d so auch alle Pfade zu weiter links liegenden Blättern.

H3 Die Beschriftungen der Nachfolger eines Knotens sind kleiner oder gleich den Beschriftungen des Knotens.

Repräsentation von Heaps

Ein *heap* A wird im Rechner als Array A zusammen mit einer Zahl *heap-size*$[A]$ repräsentiert (und nicht als verzeigerter Baum)

- Der Eintrag 1 bildet die Wurzel des *heaps*.
- Der Elternknoten des Eintrags i ist der Eintrag $\lfloor i/2 \rfloor$.
- Die linken und rechten Nachfolger des Eintrags i sind die Einträge $2i$ und $2i+1$. Übersteigt dieser Wert die Größe *heap-size*$[A]$, so existiert der entsprechende Nachfolger nicht.

Die Eigenschaften H1 und H2 sind für ein Array von Objekten automatisch gegeben. H3 bedeutet, dass $A[i/2] \geq A[i]$.

Beispiel:

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Die Höhe eines *heaps* der Größe n ist $\Theta(\log(n))$.

NB: $\log_b$ = „Anzahl der Male, die man durch b dividieren kann, bevor man 1 erreicht.“

Prozedur HEAPIFY: Spezifikation

Spezifikation von HEAPIFY(A, i):

- Wir sagen, der Teilbaum mit Wurzel i erfülle die Heapeigenschaft wenn gilt $A[j] \leq A[j/2]$ für alle von i aus erreichbaren Knoten j.
- Vor Aufruf mögen die Teilbäume mit Wurzeln $2i$ und $2i + 1$ die Heapeigenschaft erfüllen.
- Dann erfüllt i nach Aufruf von HEAPIFY(A, i) die Heapeigenschaft.

NB Erfüllt ein Knoten die Heapeigenschaft so auch alle seine „Kinder“. Verletzt ein Knoten die Heapeigenschaft so auch alle seine „Vorfahren“.

Prozedur HEAPIFY: Implementierung

```
HEAPIFY(A, i)
1   l ← 2i
2   r ← 2i + 1
3   if l ≤ heap-size[A] and A[l] > A[i]
4       then largest ← l
5       else largest ← i
6   if r ≤ heap-size[A] and A[r] > A[largest]
7       then largest ← r
8   if largest ≠ i
9       then exchange A[i] ↔ A[largest]
10      HEAPIFY(A, largest)
```

Prozedur HEAPIFY: Laufzeitanalyse

Sei $h(i)$ die Höhe des Knotens i, also die Länge des längsten Pfades von i zu einem Blatt.

NB: $h(i) = O(\log(\mathit{heap\text{-}size}[A]))$.

Sei $T(h)$ die maximale Laufzeit von HEAPIFY(A, i) wenn $h(i) = h$.

Es gilt $T(h) = \max(T(h-1), T(h-2)) + \Theta(1)$, somit $T(h) = O(h)$.

Prozedur Build-Heap

Wir wollen die Einträge eines beliebigen Arrays so permutieren, dass ein *heap* entsteht.

```
Build-Heap(A)
1  heap-size[A] ← length[A]
2  for i ← length[A]/2 downto 1 do
3        Heapify(A, i)
4        ▷ Alle Teilbäume mit Wurzel ≥ i erfüllen die Heapeigenschaft
```

Nach Aufruf von Build-Heap(A) enthält A dieselben Einträge wie zuvor, aber nunmehr bildet A einen *heap* der Größe n.

Prozedur Build-Heap: Laufzeitanalyse

Ein *heap* der Größe n enthält maximal $\lceil n/2^{h+1} \rceil$ Knoten der Höhe h.

Die Laufzeit von Build-Heap(A) ist somit:

$$\sum_{h=0}^{\lfloor \log_2(n) \rfloor} \lceil n/2^{h+1} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$

Hier haben wir verwendet:

$$\begin{array}{rcl|l} \displaystyle\sum_{k=0}^{\infty} x^k & = & \dfrac{1}{1-x} & \dfrac{\mathrm{d}}{\mathrm{d}x} \\ \displaystyle\sum_{k=0}^{\infty} k x^{k-1} & = & \dfrac{1}{(1-x)^2} & x \\ \displaystyle\sum_{k=0}^{\infty} k x^k & = & \dfrac{x}{(1-x)^2} & \end{array}$$

Merke: Build-Heap(A) läuft in $O(\mathit{length}[A])$.

Prozedur Heap-Sort

```
Heap-Sort(A)
1  Build-Heap(A)
2  for i ← length[A] downto 2 do
3       exchange A[1] ↔ A[i]
4       heap-size[A] ← heap-size[A] − 1
5       Heapify(A, 1)
```

Laufzeit von Heap-Sort(A) mit $length[A] = n$ ist $O(n \log n)$.

Prioritätsschlangen

Eine Prioritätsschlange (*priority queue*) ist eine Datenstruktur zur Verwaltung einer Menge von Objekten, die linear geordnete Schlüssel als Attribute besitzen. Eine Prioritätsschlange unterstützt die folgenden Operationen:

- INSERT(S, x): Einfügen des Elements x in die Schlange S.
- MAXIMUM(S) liefert das (ein) Element von S mit dem größten Schlüssel.
- EXTRACT-MAX(S) liefert das (ein) Element von S mit dem größten Schlüssel und löscht es aus S.

Wird S als *heap* organisiert, so laufen alle drei Operationen jeweils in Zeit $O(\log(|S|))$, außerdem erfordert das einmalige Aufbauen eines *heaps* nur lineare Zeit.

Prozedur HEAP-INSERT

```
HEAP-INSERT(A, key)
1   heap-size[A] ← heap-size[A] + 1
2   i ← heap-size[A]
3   while i > 1 and A[i/2] < key do
4         A[i] ← A[i/2]
5         i ← i/2
5   A[i] ← key
```

Verfolgt den Pfad vom ersten freien Blatt ($heap\text{-}size + 1$) zur Wurzel bis der Platz für key gefunden ist. Laufzeit $\Theta(\log n)$.

Quicksort

```
QUICKSORT(A, p, r)
    ▷ Sortiere A[p..r]
1   if p < r then
2       q ← PARTITION(A, p, r)
3       QUICKSORT(A, p, q)
4       QUICKSORT(A, q + 1, r)
```

- Die Prozedur PARTITION(A, p, r) gruppiert die Elemente von $A[p..r]$ um und bestimmt einen Index $q \in \{p, \ldots, r-1\}$ sodass alle Elemente von $A[p..q]$ kleiner oder gleich allen Elementen von $A[q+1..r]$ sind; in Zeichen: $A[p..q] \leq A[q+1..r]$.
- Dazu setzt man $x \leftarrow A[p]$ und platziert alle Elemente $\leq x$ nach vorne und alle Elemente $\geq x$ nach hinten.
- Da ja x selbst in beide Bereiche fallen kann, ist es möglich $q < r$ zu wählen.

Prozedur Partition

```
Partition(A, p, r)
1  x ← A[p]
2  i ← p − 1
3  j ← r + 1
4  while True do
5        repeat j ← j − 1 until A[j] ≤ x
6        repeat i ← i + 1 until A[i] ≥ x
7        if i < j
8              then exchange A[i] ↔ A[j]
9              else return j
```

Beachte:

- In Zeile 4 gilt die Invariante $A[p..i] \leq x \leq A[j..r]$
- $q = r$ ist nicht möglich
- $q = r$ wäre möglich wenn man Zeile 1 durch $A[r]$ ersetzte

Laufzeit von Quicksort

Sei $n = r - p + 1$ die Größe des zu bearbeitenden Arrays.

Der Aufruf Partition(A, p, r) hat Laufzeit $\Theta(n)$.

Sei $T(n)$ die Laufzeit von Quicksort(A, p, r).

Es gilt $T(n) = T(n_1) + T(n_2) + \Theta(n)$ wobei $n_1 + n_2 = n$.

Bester Fall: $n_1, n_2 = O(n)$, z.B., $= n/2$. Dann ist $T(n) = \Theta(n \log n)$.

Schlechtester Fall: $n_1 = O(n), n_2 = O(1)$ oder umgekehrt. Dann ist $T(n) = \Theta(n^2)$.

Randomisiertes Quicksort

Der schlechteste Fall tritt tatsächlich auf wenn das Array schon mehr oder weniger sortiert ist.

Beispiel: Buchungen sind nach Eingangsdatum sortiert, sollen nach Buchungsdatum sortiert werden.

Um diesen Effekt zu vermeiden wählt man das Pivotelement zufällig:

RANDOMIZED-PARTITION(A, p, r)

1 $i \leftarrow$ RANDOM(p, r)

2 exchange$A[p] \leftrightarrow A[i]$

3 **return** PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, r)

1 **if** $p < r$ **then**

2 $\quad q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)

3 $\quad$ QUICKSORT(A, p, q)

4 $\quad$ QUICKSORT$(A, q + 1, r)$

Erwartete Laufzeit

Die Laufzeit von Randomized-Quicksort ist nunmehr eine Zufallsvariable ohne festen Wert.

Wir bestimmen ihren Erwartungswert. Wir nehmen vereinfachend an, dass alle Elemente verschieden sind.

Wie groß wird der Teil $n_1 = q - p + 1$ der Partition?

Das hängt vom Rang des Pivotelements ab.

Ist das Pivotelement das Kleinste, so ist $n_1 = 1$.

Ist das Pivotelement das Zweitkleinste, so ist $n_1 = 1$.

Ist das Pivotelement das Drittkleinste, so ist $n_1 = 2$.

Ist das Pivotelement das $n-1$kleinste, so ist $n_1 = n - 2$.

Ist das Pivotelement das Größte, so ist $n_1 = n - 1$.

Also gilt für den Erwartungswert der Laufzeit $T(n)$:

$$T(n) = \frac{1}{n}(2(T(1) + T(n-1)) + \sum_{q=2}^{n-1} T(q) + T(n-q)) + \Theta(n)$$

Explizite Bestimmung der erwarteten Laufzeit

Wir wissen bereits, dass $T(n) = O(n^2)$ somit $T(n)/n = O(n)$ und somit können die ersten beiden Summanden durch $\Theta(n)$ absorbiert werden:

$$T(n) = \frac{1}{n}(\sum_{q=2}^{n-1} T(q) + T(n-q)) + \Theta(n) = \frac{2}{n}\sum_{q=2}^{n-1} T(q) + \Theta(n)$$

Wir raten $T(n) = O(n \log n)$ und probieren durch Induktion über n zu zeigen $T(n) \leq cn \ln n$ für ein noch zu bestimmendes $c > 0$.

Sei n groß genug und fest gewählt. Es gelte $T(q) \leq cq \ln q$ für alle $q < n$.

$$T(n) \leq \frac{2c}{n}\sum_{q=2}^{n-1} q \ln q + dn$$

Es ist $\sum_{q=2}^{n-1} q \ln q \leq \int_{q=2}^{n} q \ln q \, \mathrm{d}q = [\frac{1}{2}q^2 \ln q - \frac{1}{4}q^2]_2^n \leq \frac{1}{2}n^2 \ln n - \frac{1}{4}n^2$.

Mit $c \geq 2d$ bekommen wir also $T(n) \leq cn \ln n$ somit ist $T(n) = O(n \log n)$ erwiesen.

Untere Schranke für vergleichsbasiertes Sortieren

Sei $\mathcal{A}$ irgendein Algorithmus, welcher n Objekte sortiert, indem auf die Objekte nur über binäre Vergleiche der Form $o_1 \leq o_2$? zugegriffen wird.

Also nicht durch explizites Lesen der Objekte.

Beispiel aus `stdlib.h`

```
void qsort(void A[], int size, int compare(void *,void *))
```

Wir behaupten: $\mathcal{A}$ erfordert $\Omega(n \log n)$ solche Vergleiche.

Beweis der unteren Schranke

Nehmen wir an, $\mathcal{A}$ führe V Vergleiche durch.

Welche Elemente hier verglichen werden hängt i.a. vom Ausgang vorhergehender Vergleiche ab!

Die möglichen Ausgänge dieser Vergleiche partitionieren die möglichen Eingaben in 2^V Klassen.

Eingaben, die in die gleiche Klasse fallen, werden gleich behandelt.

Jede der $n!$ Permutationen von n verschiedenen Objekten erfordert unterschiedliche Behandlung.

Also muss gelten $2^V \geq n!$ oder $V \geq \log_2 n! = \Omega(n \log n)$.

[Beweis, dass $\log_2 n! = \Omega(n \log n)$.
$\log n! = \sum_{i=1}^{n} \log_2 i \geq \frac{1}{2} \log_2(\frac{n}{2}) = \frac{1}{2}(\log_2(n) - 1) = \Omega(n \log n)$.]

Bestimmung des Maximums

Das Array A enthalte n verschiedene Zahlen.

Folgender Algorithmus bestimmt das Maximum der $A[i]$.

```
MAXIMUM(A, n)
▷ Bestimmt das größte Element von A, wenn n ≥ 0
1  kandidat ← A[1]
2  for i ← 2 to n do
▷  kandidat ≥ A[1..i − 1]
3        if A[i] > kandidat then kandidat ← A[i]
4  return kandidat
```

Die Vergleichskomplexität dieses Verfahrens beträgt $V(n) = n - 1$.

Soll heißen, $n - 1$ Größenvergleiche werden durchgeführt.

Ganz analog haben wir ein Verfahren MINIMUM, das das kleinste Element bestimmt.

Vergleichskomplexität des Maximums

Die Vergleichskomplexität des Problems „Maximumbestimmung“ ist die minimale Zahl von Vergleichen, die im schlechtesten Fall erforderlich sind, um das Maximum zu bestimmen.

Die Existenz des Verfahrens Maximum belegt $V(n) \leq n - 1$.

Es gilt tatsächlich $V(n) = n - 1$.

Vergleichskomplexität des Maximums

- Sei M die Menge der Positionen im Array, an denen aufgrund der bis dato gemachten Vergleiche noch das Maximum stehen könnte.
- Am Anfang ist $M = \{1, \ldots, n\}$. Am Ende muss $|M| = 1$ sein.
- Aus M entfernen können wir eine Position i nur dann, wenn ein Vergleich stattgefunden hat, in dem $A[i]$ das kleinere Element ist.
- Ein Vergleich entfernt also höchstens ein Element aus M.
- $n - 1$ Vergleiche sind erforderlich.
- Das gilt ganz gleich wie die Vergleiche ausgehen, also auch im besten Fall.

Maximum und Minimum gleichzeitig

Es gelte, simultan das größte und das kleinste Element in einem Array zu bestimmen.

Anwendung: Skalierung von Messwerten.

Durch Aufruf von Maximum und dann Minimum erhalten wir einen Algorithmus für dieses Problem mit Vergleichskomplexität $2n - 2$.

Somit gilt für die Vergleichskomplexität $V(n)$ des Problems „Maximum und Minimum“

$$V(n) \leq 2n - 2$$

Ist das optimal?

Maximum und Minimum gleichzeitig

```
MAXIMUM-MINIMUM(A, n)
▷ Bestimmt das Maximum und das Minimum in A[1..n]
1  for i ← 1 to ⌊n/2⌋ do
2      if A[2i − 1] < A[2i]
3          then B[i] ← A[2i − 1]; C[i] ← A[2i]
4          else C[i] ← A[2i − 1]; B[i] ← A[2i]
5  if n ungerade
6      then B[⌊n/2⌋ + 1] ← A[n]; C[⌊n/2⌋ + 1] ← A[n]
7  return (MINIMUM(B, ⌈n/2⌉), MAXIMUM(C, ⌈n/2⌉))
```

- Die Elemente werden zunächst in $\lceil n/2 \rceil$ verschiedenen Paaren verglichen. Das letzte Paar besteht aus zwei identischen Elementen, falls n ungerade.
- Das Maximum ist unter den $\lceil n/2 \rceil$ „Siegern“; diese befinden sich in C.
- Das Minimum ist unter den $\lceil n/2 \rceil$ „Verlierern“; diese befinden sich in B.

Es gilt also $V(n) \leq \lceil \frac{3n}{2} \rceil - 2$.

Maximum und Minimum gleichzeitig

Behauptung: MAXIMUM-MINIMUM ist optimal, also $V(n) = \lceil \frac{3n}{2} \rceil - 2$.

Beachte: Hier unterscheiden sich bester und schlechtester Fall. Im besten Fall können wir nämlich mit $n-1$ Vergleichen Maximum und Minimum finden:

Wir vergleichen $A[i]$ mit $A[i+1]$ für $i \leftarrow 1..n-1$. Wenn stets gilt $A[i] < A[i+1]$ dann ist $A[1]$ das Minimum und $A[n]$ das Maximum.

Aber wenn die Vergleiche anders ausgehen: Schade. . . .

Beweis, dass $V(n) = \lceil \frac{3n}{2} \rceil - 2$. Teil I

- Sei MAX, bzw. MIN, die Menge der Positionen, die nach Maßgabe der bisher stattgefundenen Vergleiche noch für das Maximum, bzw. das Minimum, in Frage kommen.
- Am Anfang ist $MAX = MIN = \{1, \ldots, n\}$.
- Am Ende muss $|MAX| = |MIN| = 1$ sein.
- Wie verringern sich $|MAX|$ und $|MIN|$?
  - Bei einem Volltreffer verringern sich sowohl $|MAX|$ als auch $|MIN|$ um eins.
  - Bei einem Treffer verringern sich entweder $|MAX|$ oder $|MIN|$ um eins.
  - Bei einem Fehlschuss passiert gar nichts.
- Wir behaupten: Es kann im schlechtesten Fall höchstens $\lfloor n/2 \rfloor$ Volltreffer geben. Soll heißen, wenn's dumm läuft, sind auch mit der besten Strategie höchstens $\lfloor n/2 \rfloor$ Volltreffer zu erzielen.

Beweis, dass es höchstens $\lfloor n/2 \rfloor$ Volltreffer gibt

- Ein Volltreffer kommt genau dann vor, wenn bei einem Vergleich das größere Element in MIN war und das kleinere in MAX.
- Liegen also i, j beide in $MAX \cap MIN$, so garantiert der Vergleich von $A[i]$ mit $A[j]$ einen Volltreffer.
- Liegt dagegen $i \in MAX$, $j \in MIN$, aber $i \notin MIN$, ...
  - ...so könnte natürlich $A[i] < A[j]$ herauskommen, was ein Volltreffer wäre.
  - Es könnte aber auch $A[i] > A[j]$ herauskommen, was einen Fehlschuss bedeutet.
  - Man beachte, dass unter den Voraussetzungen i noch nie „verloren“ hat und j noch nie „gewonnen“ hat. Somit ist der Ausgang $A[i] > A[j]$ mit den bisherigen Beobachtungen konsistent.
- Alle anderen Möglichkeiten sind zu dieser analog.

Beweis, dass $V(n) = \lceil \frac{3n}{2} \rceil - 2$. Teil II und Schluss

- Aus $MIN \cup MAX$ sind insgesamt $2n - 2$ Elemente zu eliminieren.
- Die maximal möglichen $\lfloor n/2 \rfloor$ Volltreffer eliminieren $2\lfloor n/2 \rfloor$ davon.
- Die $2n - 2 - 2\lfloor n/2 \rfloor$ übriggebliebenen können wir bestenfalls durch Treffer eliminieren, sodass
  $V(n) \geq \lfloor n/2 \rfloor + 2n - 2 - 2\lfloor n/2 \rfloor = 2n - \lfloor n/2 \rfloor - 2 = \lceil 3n/2 \rceil - 2$.
- Das fällt mit der schon bewiesenen oberen Schranke zusammen, somit gilt $V(n) = \lceil 3n/2 \rceil - 2$ *quod erat demonstrandum*.

Die Selektionsaufgabe

Die Selektionsaufgabe besteht darin, von n verschiedenen Elementen das i-kleinste (sprich: [ihtkleinste]) zu ermitteln.

Das i-kleinste Element ist dasjenige, welches nach aufsteigender Sortierung an i-ter Stelle steht.

Englisch: i kleinstes Element = ith order statistic.

Das 1-kleinste Element ist das Minimum.

Das n-kleinste Element ist das Maximum.

Das $\lfloor \frac{n+1}{2} \rfloor$-kleinste und das $\lceil \frac{n+1}{2} \rceil$-kleinste Element bezeichnet man als Median.

Ist n gerade, so gibt es zwei Mediane, ist n ungerade so gibt es nur einen.

Anwendung des Medians

Fakt: Sei $x_1, \ldots, x_n$ eine Folge von Zahlen. Der Ausdruck $S(x) = \sum_{i=1}^{n} |x - x_i|$ nimmt sein Minimum am Median der x_i an.

Beispiele

- n Messwerte x_i seien so zu interpolieren, dass die Summe der absoluten Fehler minimiert wird. Lösung: Median der x_i.
- n Städte liegen auf einer Geraden an den Positionen x_i. Ein Zentrallager sollte am Median der x_i errichtet werden um die mittlere Wegstrecke zu minimieren (unter der Annahme, dass jede Stadt gleich oft angefahren wird.)
- Analoges gilt auch in 2D bei Zugrundelegung der Manhattandistanz.

Vergleichskomplexität der Selektionsaufgabe

- Durch Sortieren kann die Selektionsaufgabe mit Vergleichskomplexität $\Theta(n \log n)$ gelöst werden, somit gilt für die Vergleichskomplexität $V(n)$ der Selektionsaufgabe: $V(n) = O(n \log n)$.
- $V(n) = \Omega(n)$ ergibt sich wie beim Maximum. Mit weniger als $n - 1$ Vergleichen kann ein Element nicht als das i-kleinste bestätigt werden.
- Tatsächlich hat man $V(n) = \Theta(n)$.

Selektion mit mittlerer Laufzeit $\Theta(n)$

RANDOMIZED-SELECT(A, p, r, i)

▷ Bestimmt den Index des i-kleinsten Elements in $A[p..r]$

1 **if** $p = r$ **then return** p

2 $q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)

3 $k \leftarrow q - p + 1$

4 **if** $i \leq k$

5 **then return** RANDOMIZED-SELECT(A, p, q, i)

6 **else return** RANDOMIZED-SELECT$(A, q + 1, r, i - k)$

Laufzeit (und Vergleichskomplexität) im schlechtesten Falle: $\Theta(n^2)$.

Mittlere Laufzeit von Randomized-Select

Für den Erwartungswert $V(n)$ der Laufzeit von Randomized-Select(A, p, r, i), wobei $n = r - p + 1$, gilt die Rekurrenz:

$$T(n) \leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n)$$

Diese Rekurrenz hat die Lösung $T(n) = O(n)$ wie man durch Einsetzen und Induktion bestätigt.

Lineare Laufzeit im schlechtesten Fall

```
SELECT(A, p, r, i)
▷   Bestimmt den Index des i-kleinsten Elements in A[p..r]
1   if p = r then return p
2   Teile die A[i] in Fünfergruppen auf (plus eventuell eine kleinere Gruppe)
3   Bestimme den Median jeder Gruppe durch festverdrahtete Vergleiche
4   Bestimme durch rekursiven Aufruf von SELECT den Median dieser Mediane.
5   Vertausche in A diesen Median mit A[p]
6   q ← PARTITION(A, p, r)
7   k ← q − p + 1
8   if i ≤ k
9       then return SELECT(A, p, q, i)
10      else return SELECT(A, q + 1, r, i − k)
```

Worst-case Laufzeit von Select

Sei $T(n)$ die *worst case* Laufzeit von Select.

- Gruppenbildung und individuelle Mediane: $O(n)$.
- Bestimmung des Medians der Mediane: $T(n/5)$.
- Der Median der Mediane liegt oberhalb und unterhalb von jeweils mindestens $\frac{3n}{10}$ Elementen.
- Die größere der beiden „Partitionen“ hat also weniger als $\frac{7}{10}$ Elemente.
- Der rekursive Aufruf auf einer der beiden „Partitionen“ erfordert also $T(\frac{7n}{10})$.

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

Die Lösung ist $T(n) = O(n)$ wie man durch Einsetzen bestätigt.

NB Die Lösung von $T(n) = T(n/5) + T(8n/10) + O(n)$ ist $O(n \log n)$.

Dynamische Mengen

Eine dynamische Menge ist eine Datenstruktur, die Objekte verwaltet, welche einen Schlüssel tragen, und zumindest die folgenden Operationen unterstützt:

- SEARCH(S, k): liefert (einen Zeiger auf) ein Element in S mit Schlüssel k falls ein solches existiert; NIL sonst.
- INSERT(S, x): fügt das Element (bezeichnet durch Zeiger) x in die Menge S ein.

Oft werden weitere Operationen unterstützt, wie etwa

- DELETE(S, x): löscht das Element (bezeichnet durch Zeiger) x.
- Maximum, Minimum, Sortieren, etc. bei geordneten Schlüsseln.

Typische Anwendung: Symboltabelle in einem Compiler. Schlüssel = Bezeichner, Objekte = (Typ, Adresse, Größe, ...)

Direkte Adressierung

- Sind die Schlüssel ganze Zahlen im Bereich $1 \ldots N$, so kann eine dynamische Menge durch ein Array A der Größe N implementiert werden.
- Der Eintrag $A[k]$ ist x falls ein Element x mit Schlüssel x eingetragen wurde.
- Der Eintrag $A[k]$ ist NIL, falls die dynamische Menge kein Element mit Schlüssel k enthält.
- Die Operationen SEARCH, INSERT, DELETE werden unterstützt und haben Laufzeit $\Theta(1)$.
- Nachteile: Enormer Speicherplatz bei großem N. Nicht möglich, falls keine obere Schranke an Schlüssel vorliegt.

Hash-Tabelle

- Sei U die Menge der Schlüssel.
- Gegeben eine Funktion $h : U \to \{1, 2, 3, \ldots, n\}$, die „Hashfunktion“.
- Die dynamische Menge wird implementiert durch ein Array der Größe n.
- Das Element mit Schlüssel k wird an der Stelle $A[h(k)]$ abgespeichert.

```
SEARCH(A, k)
1  return A[h(k)]
INSERT(A, x)
1  A[h(key[x])] ← x
DELETE(A, k)
1  A[h(k)] ← NIL
```

- Zum Beispiel: $U = \mathbb{N}$, $h(k) = (k \bmod n) + 1$.
- Problem: $h(k_1) = h(k_2)$ obwohl $k_1 \neq k_2$ (Kollision). Kommt es zu Kollisionen so ist dieses Verfahren inkorrekt (also gar kein Verfahren!).

Häufigkeit von Kollisionen

- Alle n Hashwerte seien gleichwahrscheinlich.
- Die Wahrscheinlichkeit, dass k zufällig gewählte Schlüssel paarweise verschiedene Hashwerte haben ist dann:

$$\begin{aligned} & 1 \cdot \tfrac{n-1}{n} \cdot \tfrac{n-2}{n} \cdots \tfrac{n-k+1}{n} = \textstyle\prod_{i=0}^{k-1}(1 - \tfrac{i}{n}) \\ \leq & \quad \textstyle\prod_{i=0}^{k-1} e^{-i/n} \\ = & \quad e^{-\sum_{i=0}^{k-1} i/n} = e^{-k(k-1)/2n} \end{aligned}$$

- Diese Wahrscheinlichkeit wird kleiner als 50% wenn $k \geq 1 + \frac{1}{2}\sqrt{1 + 8n \ln 2}$.
- Beispiel $n = 365$, $h(k) =$„Geburtstag von k“. Bei mehr als 23 Menschen ist es wahrscheinlicher, dass zwei am selben Tag Geburtstag haben, als umgekehrt.
- Kollisionen sind häufiger als man denkt.

Kollisionsauflösung durch Verkettung

Um Kollisionen zu begegnen, hält man in jeder Arrayposition eine verkettete Liste von Objekten.

- Suchen geschieht durch Suchen in der jeweiligen Liste,
- Einfügen geschieht durch Anhängen an die jeweilige Liste,
- Löschen geschieht durch Entfernen aus der jeweiligen Liste.

SEARCH(A, k)

1 Suche in der Liste $A[h(k)]$ nach Element mit Schlüssel k

INSERT(A, x)

1 Hänge x am Anfang der Liste $A[h(k)]$ ein.

DELETE(A, k)

1 Entferne das Objekt mit Schlüssel k aus der Liste $A[h(k)]$.

Leider ist die Laufzeit jetzt nicht mehr $\Theta(1)$.

Lastfaktor

Die Hashtabelle habe m Plätze und enthalte n Einträge.

Der Quotient $\alpha := n/m$ heißt Lastfaktor.

Beachte: $\alpha > 1$ ist möglich.

Der Lastfaktor heißt auch Belegungsfaktor

Eine Hashtabelle heißt auch Streuspeichertabelle.

Vgl. Cruise Missile vs. Marschflugkörper.

Analyse von Hashing mit Verkettung

Die Hashwerte seien wiederum uniform verteilt (*simple uniform hashing*).

Dann werden die Listen im Mittel Länge α besitzen.

Die Operationen SEARCH, INSERT, DELETE haben also jeweils erwartete (=mittlere) Laufzeit

$$T \leq c(1 + \alpha)$$

für geeignete Konstanten $c > 0$.

Der Summand „1“ bezeichnet den Aufwand für das Berechnen der Hashfunktion und die Indizierung.

Der Summand α bezeichnet die lineare Laufzeit des Durchsuchens einer verketteten Liste.

Cormen *et al.* schreiben $T = O(1 + \alpha)$. Das ist m.E. nicht ganz korrekt, da nicht klar ist, von welcher Variable α abhängt (m?, n?, beides?, keins von beiden?).

Hashfunktionen

Seien die einzutragenden Objekte x irgendwie zufällig verteilt.

Die Hashfunktion sollte so beschaffen sein, dass die Zufallsvariable $h(key[x])$ uniform verteilt ist (da ja sonst manche slots leer bleiben, während andere überfüllt sind.)

Sind z.B. die Schlüssel in $\{1, \ldots, N\}$ uniform verteilt, so ist $h(x) = (x \bmod m) + 1$ eine gute Hashfunktion.

Sind z.B. die Schlüssel in $[0, 1]$ uniform verteilt, so ist $h(x) = \lceil nx \rceil$ eine gute Hashfunktion.

n wie immer die Größe der Hashtabelle.

Die Schlüssel sind meist nicht uniform verteilt:

Bezeichner in einer Programmiersprache: `count`, `i`, `max_zahl` häufiger als `zu6fgp98qq`. Wenn `kli` dann oft auch `kli1`, `kli2`, etc.

Nichtnumerische Schlüssel

. . . müssen vor Anwendung einer „Lehrbuch-Hashfunktion“ zunächst in Zahlen konvertiert werden.

Zeichenketten etwa unter Verwendung der Basis 256:

`'p'` $= 112$, `'q'` $= 116$, also `"pq"` $= 112 \cdot 256 + 116 = 28788$.

Divisionsmethode

Wie immer: Schlüssel: $1 \ldots N$, Hashwerte: $1 \ldots m$.

Hashfunktion: $h(k) = k \bmod m$.

- m sollte keine Zweierpotenz sein, da sonst $h(k)$ nicht von allen Bits (von k abhängt.
- Ist k eine Kodierung eines Strings im 256er System, so bildet h bei $m = 2^p - 1$ zwei Strings, die sich nur durch eine Transposition unterscheiden, auf denselben Wert ab.
- Eine gute Wahl für m ist eine Primzahl die nicht nahe bei einer Zweierpotenz liegt. Z.B. $n = 2000$, vorauss. Lastfaktor $\alpha = 3$: Tabellengröße $m = 701$ bietet sich an.
- Bei professionellen Anwendungen empfiehlt sich ein Test mit „realen Daten“.

Multiplikationsmethode

Hashfunktion: $h(k) = \lfloor m(kA \bmod 1 \rfloor$ für $A \in]0, 1[$.

Hier $x \bmod 1$ =„gebrochener Teil von x“, z.B.: $\pi \bmod 1 = 0,14159\ldots$.

Rationale Zahlen A mit kleinem Nenner führen zu Ungleichverteilungen, daher empfiehlt sich die Wahl $A = (\sqrt{5} - 1)/2$ („Goldener Schnitt“).

Vorteile der Multiplikationsmethode:

- Arithmetische Progressionen von Schlüsseln $k = k_0, k_0 + d, k_0 + 2d, k_0 + 3d, \ldots$ werden ebenmäßig verstreut.
- Leicht zu implementieren wenn $m = 2^p$ (hier unproblematisch) und $N < 2^w$, wobei w die Wortlänge ist: Multipliziere k mit $\lfloor A \cdot 2^w \rfloor$. Dies ergibt zwei w-bit Wörter. Vom Niederwertigen der beiden Wörter bilden die p Höchstwertigen Bits den Hashwert $h(k)$.

Weiterführendes

- Universelles Hashing: Zufällige Wahl der Hashfunktion bei Initialisierung der Tabelle, dadurch Vermeidung systematischer Kollisionen, z.B. Provokation schlechter Laufzeit durch bösartig konstruierte Benchmarks.
- Gute Hashfunktionen können zur Authentizierung verwendet werden, z.B., MD5 *message digest*.

Offene Adressierung

Man kann auf verkettete Listen verzichten, indem man bei Auftreten einer Kollision eine andere Arrayposition benutzt.

Dazu braucht man eine zweistellige Hashfunktion h : Schlüssel $\times \mathbb{N} \to$ Hashwerte.

```
INSERT(T, x)
1  i ← 1
2  while i ≤ m and h(key[x], i) ≠ NIL do
3        i ← i + 1
4  if i ≤ m
5        then T[h(key[x], i)] = x
6        else error "hash table overflow"
```

Für jeden Schlüssel k sollte die Probierfolge (*probe sequence*)

$$h(k,1), h(k,2), h(k,3), \ldots, h(k,m)$$

eine Permutation von $1, 2, 3, \ldots, m$ sein, damit jede Position irgendwann probiert wird.

Offene Adressierung

```
SEARCH(T, k)
1  i ← 0
2  repeat
3        i ← i + 1; j ← h(k, i)
4  until i > m or T[j] = NIL or key[T[j]] = k
5  if i ≤ m and key[T[j]] = k
6        then return T[j]
7        else return NIL
```

NB: Tabelleneinträge sind Objekte zuzüglich des speziellen Wertes NIL.

Z.B. Zeiger auf Objekte oder Nullzeiger.

Einschränkung: Bei offener Adressierung ist Löschen nur schwer zu realisieren.

Hashfunktionen für offene Adressierung

- Lineares Probieren (*linear probing*):

  $$h(k,i) = (h'(k) + i) \bmod m$$

  Problem: Lange zusammenhängende Blöcke besetzter Plätze entstehen (*primary clustering*), dadurch oft lange Probierdauer.

- Quadratisches Probieren (*quadratic probing*):

  $$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

Quadratisches Probieren ist wesentlich besser als lineares Probieren, aber beide Verfahren haben folgenden Nachteil:

Wenn $h(k_1, 0) = h(k_2, 0)$, dann $h(k_1, i) = h(k_2, i)$, d.h., kollidierende Schlüssel haben dieselbe Probiersequenz.

Insgesamt gibt es nur m verschiedene Probiersequenzen (von $m!$ Möglichen!); das führt auch zu Clusterbildung (*secondary clustering*).

Hashfunktionen für offene Adressierung

- Double hashing

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

Jede Probiersequenz ist eine arithmetische Progression, Startwert und Schrittweite sind durch Hashfunktionen bestimmt.

Damit alle Positionen probiert werden, muss natürlich $\mathrm{ggT}(h_2(k), m) = 1$ sein. Z.B. m Zweierpotenz und $h_2(k)$ immer ungerade.

Es gibt dann $\Theta(m^2)$ Probiersequenzen.

Analyse der Offenen Adressierung

Vereinfachende Annahme: Die Schlüssel seien so verteilt, dass jede der $m!$ Probiersequenzen gleichwahrscheinlich ist (*uniform hashing*).

Diese Annahme wird durch *double hashing* approximiert aber nicht erreicht.

Satz: In einer offen adressierten Hashtabelle mit Lastfaktor $\alpha = n/m < 1$ ist die zu erwartetende Dauer einer erfolglosen Suche beschränkt durch $1/(1-\alpha)$.

Beispiel: Lastfaktor $\alpha = 0,9$ (Tabelle zu neunzig Prozent gefüllt): Eine erfolglose Suche erfordert im Mittel weniger als 10 Versuche (unabhängig von m, n).

Bemerkung: Dies ist auch die erwartete Laufzeit für eine Insertion.

Beweis des Satzes

Sei X eine Zufallsvariable mit Werten aus $\mathbb{N}$.

Dann ist

$$E[X] := \sum_{i=0}^{\infty} i\Pr\{X = i\} = \sum_{i=1}^{\infty} \Pr\{X \geq i\}$$

Dies deshalb, weil $\Pr\{X \geq i\} = \sum_{j=i}^{\infty} \Pr\{X = j\}$.

Daher ergibt sich für die erwartete Suchdauer D:

$$D = \sum_{i=1}^{\infty} \Pr\{\text{„Mehr als } i \text{ Versuche finden statt“}\}$$

$$\Pr\{\text{„Mehr als } i \text{ Versuche finden statt“}\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} \leq \alpha^i$$

Also, $D \leq \sum_{i=1}^{\infty} \alpha^i = 1/(1-\alpha)$.

Analyse der Offenen Adressierung

Satz: In einer offen adressierten Hashtabelle mit Lastfaktor $\alpha = n/m < 1$ ist die zu erwartetende Dauer einer erfolgreichen Suche beschränkt durch $(1 - \ln(1 - \alpha))/\alpha$.

Beispiel: Lastfaktor $\alpha = 0,9$: Eine erfolgreiche Suche erfordert im Mittel weniger als 3,67 Versuche (unabhängig von m, n).

Lastfaktor $\alpha = 0,5$: mittlere Suchdauer $\leq 3,39$.

Achtung: All das gilt natürlich nur unter der idealisierenden Annahme von *uniform hashing*.

Beweis

Die beim Aufsuchen des Schlüssels durchlaufene Probiersequenz ist dieselbe wie die beim Einfügen durchlaufene.

Die Länge dieser Sequenz für den als $i+1$-ter eingefügten Schlüssel ist im Mittel beschränkt durch $1/(1-i/m) = m/(m-i)$. (Wg. vorherigen Satzes!)

Gemittelt über alle Schlüssel, die eingefügt wurden, erhält man also

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{1}{\alpha}(H_m - H_{m-n})$$

als obere Schranke für den Erwartungswert der Suchdauer.

Hier ist $H_k = \sum_{i=1}^{k} 1/i$ die k-te harmonische Zahl. Es gilt mit Integralabschätzung $\ln k \leq H_k \leq 1 + \ln k$.

Daraus ergibt sich der Satz.

Zusammenfassung: Hashing

- Hashing = Speichern von Objekten an Arraypositionen, die aus ihrem Schlüssel berechnet werden.
- Die Zahl der Arraypositionen ist i.a. wesentlich kleiner als die der Schlüssel.
- Kollisionsauflösung durch Verkettung: Jede Arrayposition enthält eine verkettete Liste.
- Offene Adressierung: Bei Kollision wird eine andere Arrayposition probiert.
- Hashfunktionen für einfaches Hashing: Multiplikationsmethode, Divisionsmethode.
- Hashfunktionen für offene Adressierung: Lineares, quadratisches Probieren. Double Hashing.
- Analyse unter Uniformitätsannahme, Komplexität jeweils als Funktion des Lastfaktors α (Auslastungsgrad):
  - Suchen und Einfügen bei einfachem Hashing: $c(1+\alpha)$
  - Einfügen und erfolgloses Suchen bei offener Adressierung: $1/(1-\alpha)$
  - Erfolgreiches Suchen bei offener Adressierung: $(1-\ln(1-\alpha))/\alpha$

Schlussbemerkung

Hashing realisiert exzellente mittlere Laufzeiten. Das *worst case* Verhalten ist aber extrem schlecht.

Für sicherheitskritische Anwendungen (Flugüberwachung) empfehlen sich andere Datenstrukturen, wie binäre Suchbäume.

Siehe Vorlesung am kommenden Freitag.