Hashverfahren

Dank an: Beate Bollig, TU Dortmund!

Hashing

- Hashverfahren
  - Aufgabe
  - Realisierung
- Hashfunktionen
  - Anforderungen
  - Wahl einer Hashfunktion
- Kollisionen
  - Geburtstagsparadoxon
  - Strategien zur Kollisionsbehandlung
  - Hashverfahren mit Verkettung der Überläufer
  - Offene Hashverfahren
- Ausblick
  - Universelles Hashing

Hashing

Aufgabe

Dynamische Verwaltung von Daten, wobei jeder Datensatz eindeutig durch einen Schlüssel charakterisiert ist

Viele Anwendungen benötigen nur einfache Daten-Zugriffsmechanismen (dictionary operations):

- Suche nach Datensatz bei gegebenem Schlüssel x
  search(x)
- Einfügen eines neuen Datensatzes d mit Schlüssel x
  insert(x, d) (abgekürzt insert(x))
- Entfernen eines Datensatzes bei gegebenem Schlüssel x
  delete(x)

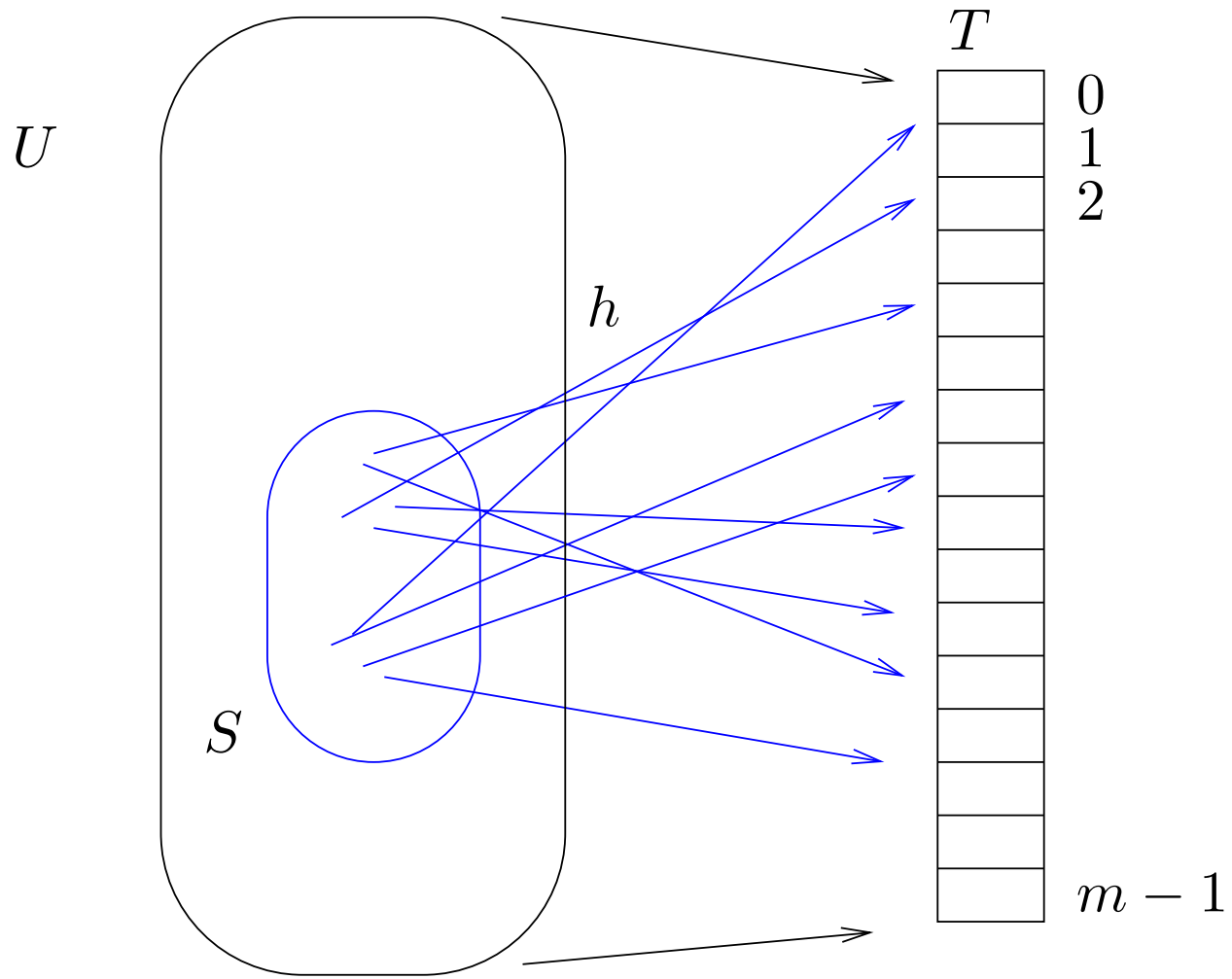
Menge potentieller Schlüssel (Universum) kann sehr groß sein!

Hashing

Hashing

- Menge U potentieller Schlüssel sehr groß, aktuelle Schlüsselmenge S jeweils nur kleine Teilmenge des Universums (im allgemeinen S nicht bekannt)
- Idee: durch Berechnung feststellen, wo Datensatz mit Schlüssel x gespeichert
- Abspeicherung der Datensätze in einem Array T mit Indizes $\{0, 1, \ldots, m-1\}$: Hashtabelle
- Hashfunktion h liefert für jeden Schlüssel $x \in U$ eine Adresse in Hashtabelle, d.h. $h : U \to \{0, 1, \ldots, m-1\}$.

Hashing



Vorteil von Hashing

Sei $n := |S|$.

- Balancierte Suchbäume (AVL-Bäume, B-Bäume): dictionary operations Komplexität $O(\log n)$
- Hashing:
  für alle Operationen mittlere Komplexität $O(1)$

Belegungsfaktor

Quotient $\beta := n/m$ heißt Belegungsfaktor oder Auslastungsfaktor einer Hashtabelle der Größe m.

Mittlerer Aufwand für dictionary operations als Funktion in β abschätzbar

$\rightarrow$ Anzahl aktueller Schlüssel geht nur indirekt in Aufwand ein

Hashing

Herausforderung:

Anzahl möglicher Schlüssel viel größer als Hashtabelle, also $|U| >> m$

- Hashfunktion muss verschiedene Schlüssel x_1 und x_2 auf gleiche Adresse abbilden.
- x_1 und x_2 beide in aktueller Schlüsselmenge
  $\longrightarrow$ Adresskollision

Hashverfahren gegeben durch:

- eine Hashtabelle,
- eine Hashfunktion, die Universum der möglichen Schlüssel auf Adressen einer Hashtabelle abbildet,
- eine Strategie zur Auflösung möglicher Adresskollisionen.

Anforderungen an Hashfunktionen

Gute Hashfunktionen sollten:

- surjektiv sein, d.h. den ganzen Wertebereich umfassen,
- die zu speichernden Schlüssel (möglichst) gleichmäßig verteilen, d.h. für alle Speicherplätze i und j sollte gelten $|h^{-1}(i)| \approx |h^{-1}(j)|$,
- effizient berechenbar sein.

Voraussetzung:

$U = \{0, 1, \ldots, N-1\}$ und

$h : U \to \{0, 1, \ldots, m-1\}$

Häufig: Divisions- oder Kongruenzmethode

$h(x) :\equiv x \bmod m \to |h^{-1}(i)| \in \{\lfloor N/m \rfloor, \lceil N/m \rceil\}$

Wahl einer Hashfunktion

$h(x) :\equiv x \bmod m$

Problem: Daten oft nicht gleichverteilt!

Beispiel: Texte in Zahlen übertragen, oft viele Leerzeichen, bestimmte Wörter häufiger etc.

Wichtig: geeignete Wahl von m

- m Zweierpotenz: $x \bmod m$ wählt nur die letzten $\log m$ Bits
- m Primzahl: $x \bmod m$ *beeinflusst* alle Bits

Beispiel: $m = 11$ und $S = \{49, 22, 6, 52, 76, 34, 13, 29\}$

Hashwerte: $h(49) = 49 \bmod 11 = 5$, $h(22) = 22 \bmod 11 = 0$, 6, 8, 10, 1, 2, 7

Erinnerung:

Im allgemeinen unvermeidbar, dass Kollisionen auftreten, denn aus $N >> m$ folgt Existenz eines Speicherplatzes i mit $|h^{-1}(i)| \geq N/m$.

Frage:

Sei $n := |S|$. Wie wahrscheinlich sind Kollisionen bei $n << m$?

Geburtstagsparadoxon

Bei wie vielen (zufällig gewählten) Personen ist es *wahrscheinlich*, dass hiervon zwei am selben Tag (und Monat) Geburtstag haben?

Annahme:

- Daten unabhängig
- Prob($h(x) = j$)=$1/m$

Prob(i-tes Datum kollidiert nicht mit den ersten $i-1$ Daten, wenn diese kollisionsfrei sind)= $\frac{m-(i-1)}{m}$

Intuition:

Egal welche Speicherplätze die ersten $i-1$ Daten belegen, $m-i+1$ der m Möglichkeiten sind *gut*.

Prob(n Daten kollisionsfrei) = $\frac{m-1}{m} \cdot \frac{m-2}{m} \ldots \frac{m-n+1}{m}$

Beispiel: $m = 365$

Prob(23 Daten kollisionsfrei) $\approx$ 0.49

Prob(50 Daten kollisionsfrei) $\approx$ 0.03

Prob($2m^{1/2}$ Daten kollisionsfrei) =

$$\underbrace{\frac{m-1}{m}\cdots}\underbrace{\frac{m-m^{1/2}}{m}\cdots\frac{m-2m^{1/2}+1}{m}}$$

$$\leq \quad 1 \quad \cdot \quad \left(\frac{m-m^{1/2}}{m}\right)^{m^{1/2}} \quad = \left(1-\frac{1}{m^{1/2}}\right)^{m^{1/2}} \approx \frac{1}{e}$$

Hashing muss mit Kollisionen leben und benötigt Strategien zur Kollisionsbehandlung.

Kollisionsbehandlung

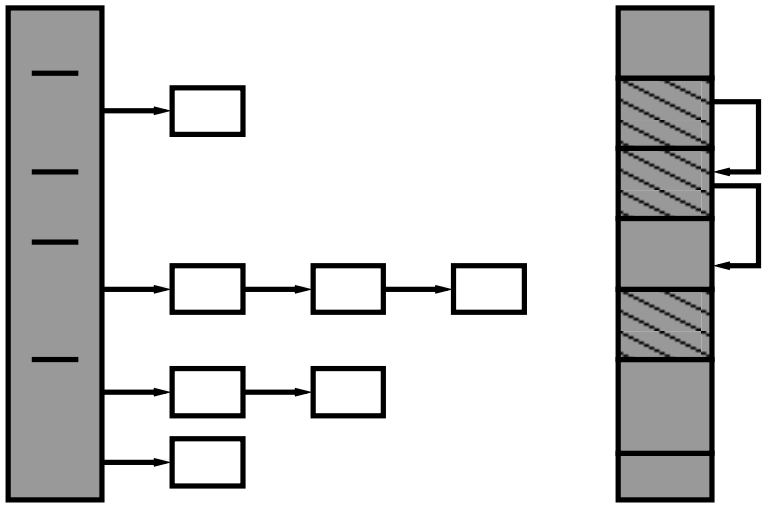
Hashverfahren unterscheiden sich in Strategien zur Kollisionsbehandlung:

- mittels verketteter Listen:
  Jede Komponente der Hashtabelle enthält Zeiger auf Überlaufliste.
- mittels offener Adressierung:
  Im Kollisionsfall nach fester Regel alternativen freien Platz in Hashtabelle suchen. Für jeden Schlüssel Reihenfolge, in der Speicherplätze betrachtet werden, vorgegeben:
  Sondierungsfolge

Kollisionsbehandlung

Verschiedene Arten der Kollisionsbehandlung:

- mittels verketteter Listen (links)
- mittels offener Adressierung (rechts)



Hashing mit Verkettung

Realisierung:

Jede Komponente der Hashtabelle enthält Zeiger auf paarweise disjunkte lineare Listen. Die i-te Liste $L(i)$ enthält alle Schlüssel $x \in S$ mit $h(x) = i$.

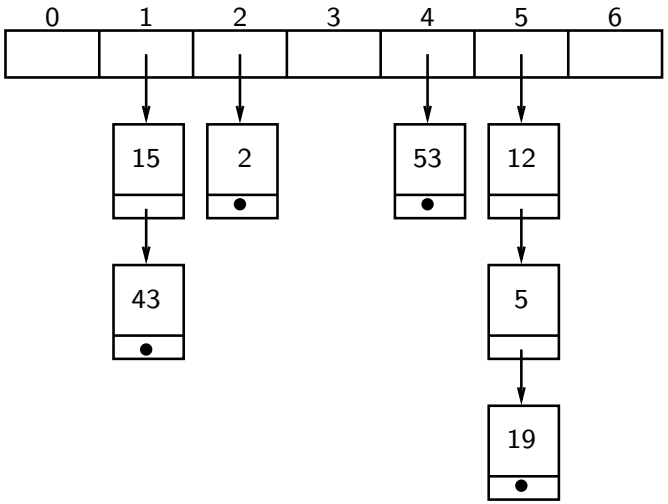
Vorteil: Alle Operationen werden unterstützt und
$n > m$ möglich (für $n >> m$ jedoch Rehashing ratsam).
Nachteil: Speicherplatzbedarf für Zeiger

- search(x): Berechne $h(x)$ und suche in Liste $L(h(x))$.
- insert(x) (nach erfolgloser Suche): Berechne $h(x)$ und füge x in Liste $L(h(x))$ ein.
- delete(x) (nach erfolgreicher Suche): Berechne $h(x)$, suche x in Liste $L(h(x))$ und entferne x.

Beispiel Verkettung der Überläufer

Beispiel: $m = 7$ und $h(x) = x \bmod m$

$$S = \{2, 5, 12, 15, 19, 43, 53\}$$



Analyse

Bei zufälligen Daten und ideal streuenden Hashfunktion gilt für

$$X_{ij} := \begin{cases} 1 & i\text{-tes Datum kommt in Liste } L(j) \\ 0 & \text{sonst} \end{cases}$$

$\text{Prob}(X_{ij} = 1) = \frac{1}{m}$

$\rightarrow \text{E}(X_{ij}) = 1 \cdot \frac{1}{m} + 0 \cdot \frac{m-1}{m} = \frac{1}{m}$

$X_j = X_{1j} + \cdots + X_{nj}$ zählt Anzahl Daten in Liste $L(j)$.

$\text{E}(X_j) = \text{E}(X_{1j} + \cdots + X_{nj}) = \text{E}(X_{1j}) + \cdots + \text{E}(X_{nj}) = \frac{n}{m}$

Analyse

- Erfolglose Suche in Liste $L(j)$:

  Inklusive nil-Zeiger durchschnittlich $1 + \frac{n}{m} = 1 + \beta$ Objekte betrachten
  Beispiel: Für $n \approx 0.95 \cdot m$, ist dies ≈ 1.95.

- Erfolgreiche Suche in Liste $L(j)$: der Länge ℓ

  Jede Position in der Liste hat Wahrscheinlichkeit $1/\ell$, also $\frac{1}{\ell}(1 + 2 + \cdots + \ell) = \frac{\ell+1}{2}$.
  Durchschnittliche Listenlänge hier: $1 + \frac{n-1}{m}$
  (Liste enthält sicher das gesuchte Datum und die anderen $n - 1$ Daten sind zufällig verteilt.)
  Also erwartete Suchdauer $\frac{1}{2}(1 + \frac{n-1}{m} + 1) = 1 + \frac{n-1}{2m} \approx 1 + \frac{\beta}{2}$
  Beispiel: Für $n \approx 0.95 \cdot m$, ist dies ≈ 1.475.

Hashverfahren mit offener Adressierung

Erinnerung:

Im Kollisionsfall nach fester Regel alternativen freien Platz in Hashtabelle suchen (Sondierungsfolge).

Voraussetzung: Auswertung von h gilt als eine Operation.

$t(i, j)$:= Position des i-ten Versuchs zum Einfügen von Daten x mit $h(x) = j$

Anforderung an Funktion t:

- auch t in Zeit $O(1)$ berechenbar
- $t(0, j) = j$
- $t(\cdot, j) : \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$ bijektiv

Operationen bei offener Adressierung

- search(x):
  - Berechne $j := h(x)$.
  - Suche x an den Positionen $t(0, j), \ldots, t(m-1, j)$.
  - Abbruch, wenn x gefunden oder freie Stelle entdeckt (kein Datum mit Schlüssel x).
- insert(x) nach erfolgloser Suche:
  Freien Platz finden (sonst Overflow) und
  x dort einfügen.
- delete(x) nach erfolgreicher Suche:
  Das Datum kann nicht einfach entfernt werden, da search frühzeitig Lücken finden würde und eine Suche fälschlicherweise als erfolglos abbrechen könnte.

Entfernen bei offener Adressierung

Problem: Datum kann bei Operation delete(x) nicht ohne weiteres gelöscht werden.

Ausweg: Speicherplatz/Position als besetzt, noch nie besetzt oder wieder frei markieren.

$\rightarrow$ Suche wird nur an Positionen mit Markierung noch nie besetzt vorzeitig abgebrochen.

Problem: Im Laufe der Zeit keine Position mehr, die mit noch nie besetzt markiert ist.

$\rightarrow$ Hashing wird ineffizient.

Offenes Hashing nur bei Anwendungen mit search und insert.

Strategien zur Kollisionsbehandlung

- Lineares Sondieren
- Quadratisches Sondieren
- Multiplikatives Sondieren
- Doppeltes Hashing

Hilfsmittel bei der Analyse: ideales Hashing

Lineares Sondieren

$t(i, j) := (i + j) \bmod m$

Beispiel: $m = 19$ und $j = h(x) = 7$

Sondierungsfolge:
7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 0, 1, 2, 3, 4, 5, 6

Problem: *Clusterbildung*
Tendenz, dass immer längere zusammenhängende, belegte Abschnitte in der Hashtabelle entstehen, sogenannte *Cluster*
$\rightarrow$ erhöhte Suchzeiten

Clusterbildung

Beispiel: $m = 19$, Positionen $2, 5, 6, 9, 10, 11, 12, 17$ belegt

$h(x)$:	0	1	2	3	4	5	6	7	8	9	10
landet an Position:	0	1	3	3	4	7	7	7	8	13	13
W.keit:	$\frac{1}{19}$	$\frac{1}{19}$	0	$\frac{2}{19}$	$\frac{1}{19}$	0	0	$\frac{3}{19}$	$\frac{1}{19}$	0	0

$h(x)$:	11	12	13	14	15	16	17	18
landet an Position:	13	13	13	14	15	16	18	18
W.keit:	0	0	$\frac{5}{19}$	$\frac{1}{19}$	$\frac{1}{19}$	$\frac{1}{19}$	0	$\frac{2}{19}$

Ideales Hashing

Wunsch:

Zu jedem Zeitpunkt alle Positionen die gleiche Wahrscheinlichkeit besetzt zu werden

Beobachtung:

Das geht numerisch nicht genau, im Beispiel 11 freie Plätze, 19 mögliche Hashwerte, also alle Wahrscheinlichkeiten $k/19$, nicht $k/11$

Modell des idealen Hashings:

Alle $\binom{m}{n}$ Möglichkeiten, die n besetzten Plätze für n Schlüssel auszuwählen, haben die gleiche Wahrscheinlichkeit.

Lineares Sondieren weit vom idealen Hashing entfernt.

Quadratisches Sondieren

$t(i,j) := j + (-1)^{i+1} \cdot \lfloor \frac{i+1}{2} \rfloor^2 \bmod m$

Sondierungsfolge:

$j,\ j+1^2,\ j-1^2,\ j+2^2,\ j-2^2, \ldots,\ j+(\frac{m-1}{2})^2,\ j-(\frac{m-1}{2})^2$

Beispiel: $m = 19$ und $j = h(x) = 7$

Sondierungsfolge:

$$7,\ 8,\ 6,\ 11,\ 3,\ 16,\ -2 = 17,\ 23 = 4,\ -9 = 10,\ 32 = 13,\ -18 = 1,$$

$$43 = 5,\ -29 = 9,\ 56 = 18,\ -42 = 15,\ 71 = 14,\ -57 = 0,\ 88 = 12,\ -74 = 2$$

Quadratisches Sondieren

Frage: Ist $t(\cdot, j)$ für alle j und m bijektiv?

Nein, aber immer wenn $m \equiv 3 \mod 4$ und m eine Primzahl ist

(Beweis: Zahlentheorie)

Besser als lineares Sondieren, aber für großes m sind die ersten Werte noch *nah* an j

Multiplikatives Sondieren I/II

Hier: $h(x) = x \mod (m-1) + 1$ und damit in $\{1, \ldots, m-1\}$

$t(i,j) := i \cdot j \mod m,\ 1 \leq i \leq m-1$

Hashwerte $1, \ldots, m-1$

Beispiel: $m = 19$ und $j = h(x) = 7$

Sondierungsfolge:

$i \cdot j$		7	14	21	28	35	42	49	56	63
$i \cdot j$	mod 19	7	14	2	9	16	4	11	18	6

$i \cdot j$		70	77	84	91	98	105	112	119	126
$i \cdot j$	mod 19	13	1	8	15	3	10	17	5	12

Multiplikatives Sondieren II/II

Frage: Ist $t(\cdot, j)$ für alle j und m bijektiv?

Nein, aber immer wenn m Primzahl und $j \neq 0$

Beweis: durch Widerspruch

Falls nicht, gibt es $1 \leq i_1 < i_2 \leq m-1$ mit

$$
\begin{aligned}
& i_1 \cdot j \equiv i_2 \cdot j \mod m \\
\Rightarrow \quad & j \cdot (i_2 - i_1) \equiv 0 \mod m \\
\Rightarrow \quad & j \cdot (i_2 - i_1) \text{ ist Vielfaches von } m \\
\Rightarrow \quad & \text{Primfaktorzerlegung von } j \cdot (i_2 - i_1) \text{ muss } m \text{ enthalten}
\end{aligned}
$$

Widerspruch zu $1 \leq j \leq m-1$, $1 \leq i_2 - i_1 \leq m-1$

Doppeltes Hashing I/II

Sei $h_1(x) \equiv x \mod m$ und $h_2(x) \equiv x \mod (m-2)+1$.

i-te Position für x: $h_1(x) + i \cdot h_2(x) \mod m$, $1 \leq i \leq m-1$

Beispiel: $m = 19$ und $x = 47$

$h_1(47) \equiv 47 \mod 19 = 9$ und $h_2(47) \equiv 47 \mod 17+1 = 14$

Sondierungsfolge:

$9, 4, 18, 13, 8, 3, 17, 12, 7, 2, 16, 11, 6, 1, 15, 10, 5, 0, 14$

Doppeltes Hashing II/II

Beobachtung:

$i \cdot h_2(x)$ durchläuft für $1 \leq i \leq m-1$ die Werte $1, \ldots, m-1$ in irgendeiner Reihenfolge, ergänzt wird $0 = 0 \cdot h_2(x)$

Durch den Summanden $h_1(x)$ wird der Anfang zufällig verschoben.

Doppeltes Hashing kommt dem idealen Hashing am nächsten.

Analyse

Analyse der Kollisionsstrategien schwierig

Wegen notwendiger Modellierung der Clusterbildung ist z.B: Durchschnittsanalyse für Lineares Sondieren nicht ganz einfach.

- Zeit für erfolgreiche Suche $\approx 1/2(1 + \frac{1}{1-\beta})$ mit $\beta := n/m$
- Zeit für erfolglose Suche $\approx 1/2(1 + \frac{1}{(1-\beta)^2})$

Beispiel: Auslastung 95%

- Erfolgreiche Suche: durchschnittlich ≈ 10.5 Positionen
- Erfolglose Suche: durchschnittlich ≈ 200.5 Positionen

Ideales Hashing analysieren, um Grenzen auszuloten

Ideales Hashing, erfolglose Suche

Erfolglose Suche beim idealen Hashing:

- Hashtabelle nicht voll, d. h. $m \geq n + 1$,
  n Daten abgespeichert
- $f(n, m) :=$ erwartete Suchzeit.
- mit W.keit 1 Suche an Position $t(0, h(x))$
  - Wahrscheinlichkeit $(m - n)/m$: Position leer, Restkosten 0
  - Wahrscheinlichkeit n/m: Position besetzt,
    erwartete Restkosten $f(n - 1, m - 1)$
    (Resttabelle der Länge $m - 1$ enthält $n - 1$ zufällig verteilte Daten.)

Ideales Hashing, erfolglose Suche

Also

$$
\begin{aligned}
f(n,m) &= 1 + \frac{m-n}{m} \cdot 0 + \frac{n}{m} \cdot f(n-1, m-1) \\
&= 1 + \frac{n}{m} \cdot f(n-1, m-1) \\
&= 1 + \frac{n}{m} \cdot \left(1 + \frac{n-1}{m-1} \cdot f(n-2, m-2)\right) \\
&= 1 + \frac{n}{m} + \frac{n}{M} \cdot \frac{n-1}{m-1} \cdot \left(1 + \frac{n-2}{m-2} \cdot f(n-3, m-3)\right) \\
&= 1 + \frac{n}{m} + \frac{n}{m} \cdot \frac{n-1}{m-1} + \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots
\end{aligned}
$$

Und jetzt?

Ausprobieren, raten, ...

Ideales Hashing, erfolglose Suche

$f(n,m) = \frac{m+1}{m+1-n}$ $\qquad (\approx \frac{1}{1-\beta})$

Beweis durch Induktion über n:

$n = 0$: $f(0,m) = 1$ und $\frac{m+1}{m+1-0} = 1$.

$n-1 \to n$:

$$\begin{aligned} f(n,m) &= 1 + \frac{n}{m} \cdot f(n-1, m-1) \\ &\overset{I.V.}{=} 1 + \frac{n}{m} \cdot \frac{m}{m-(n-1)} \\ &= 1 + \frac{n}{m+1-n} = \frac{m+1}{m+1-n}\,. \end{aligned}$$

Für $n \approx 0.95 \cdot (m+1)$ im Schnitt ≈ 20 Positionen
(Ersparnisfaktor 10 gegenüber linearem Sondieren)

Ideales Hashing, erfolgreiche Suche

Erfolgreiche Suche beim idealen Hashing:

- Gesuchtes Datum x wurde als k-tes Datum eingefügt.
- Erfolgreiche Suche nach x verläuft bis zum Finden von x identisch zu der erfolglosen Suche nach x direkt vor dem Einfügen von x.
- Also $k-1$ besetzte Stellen $\rightarrow$ erwartete Suchzeit $\frac{m+1}{m+1-(k-1)}$
- Wenn jedes der n Daten mit Wahrscheinlichkeit $\frac{1}{n}$ gesucht wird, beträgt die erwartete Suchzeit $\frac{1}{n} \sum_{1 \leq k \leq n} \frac{m+1}{m-k+2}$.

Ideales Hashing, erfolgreiche Suche

$$\frac{1}{n}\sum_{1\le k\le n}\frac{m+1}{m-k+2} = \frac{m+1}{n}\underbrace{\left(\frac{1}{m-n+2}+\cdots+\frac{1}{m+1}\right)}_{\sum_{1\le i\le m+1}\frac{1}{i}\;-\;\sum_{1\le i\le m-n+1}\frac{1}{i}}$$

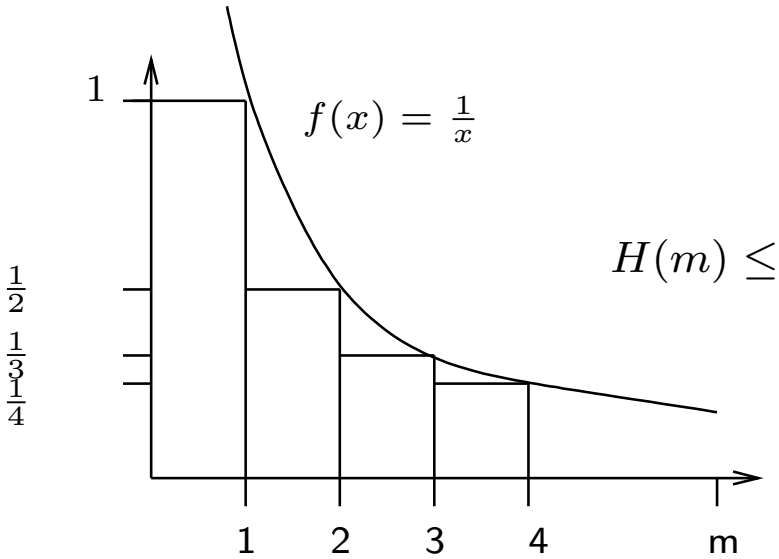
$1+\frac{1}{2}+\cdots+\frac{1}{m}$ kommt oft vor, heißt harmonische Reihe, ihr Wert wird mit $H(m)$ bezeichnet.

Frage: Wie groß ist $H(m)$ ungefähr?

$$\ln(m+1) \le H(m) \le \ln m + 1 \rightarrow H(m) \approx \ln m$$

Harmonische Reihe I/II

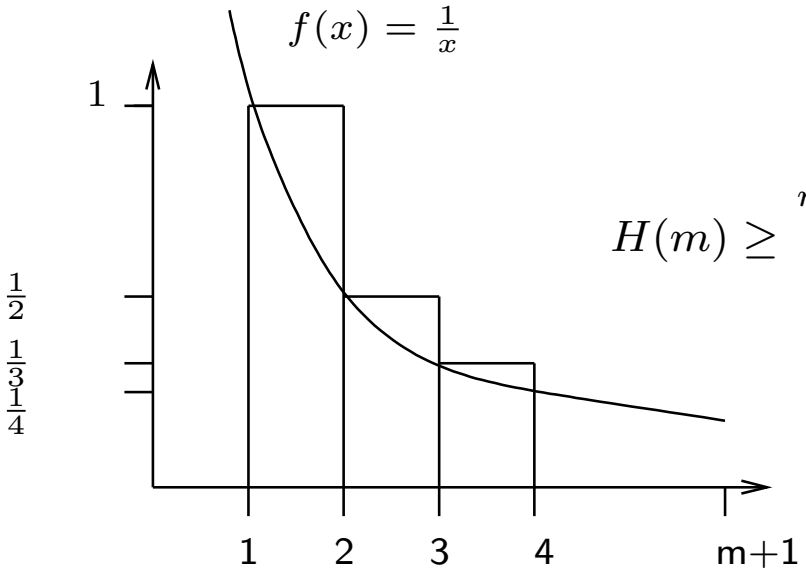
$$H(m) \leq \ln m + 1$$



$$H(m) \leq 1 + \int_1^m \frac{1}{x} dx = \ln m + 1$$

Harmonische Reihe II/II

$$\ln(m+1) \leq H(m)$$



$$H(m) \geq \int_1^{m+1} \frac{1}{x} dx = \ln(m+1)$$

$$\ln(m+1) \leq H(m) \leq \ln m + 1$$

Ideales Hashing, erfolgreiche Suche

Die erwartete Zeit einer erfolgreichen Suche beim idealen Hashing beträgt

$$\frac{m+1}{n}(H(m+1) - H(m-n+1))$$
$$\approx \frac{m+1}{n}(\ln(m+1) - \ln(m-n+1)) = \frac{m+1}{n} \ln \frac{m+1}{m+1-n}$$
$$\approx 1/\beta \ln \frac{1}{1-\beta}$$

Für $n \approx 0.95 \cdot (m+1)$ im Schnitt ≈ 3.15 Positionen
(Ersparnisfaktor größer als 3 gegenüber linearem Sondieren)

Universelles Hashing

Problem

Trotz hervorragender *Average-Case-Komplexität* von $\Theta(1)$ sehr schlechtes *Worst-Case-Verhalten* (alle aktuellen Schlüssel auf dieselbe Adresse der Hashtabelle) $\rightarrow \Theta(n)$.

Beobachtung:

Bei festgelegter Hashfunktion bestimmte Schlüsselmenge, die dieses ungünstige Verhalten hervorrufen kann.

Ausweg:

Zur Laufzeit zufällig Hashfunktion aus Klasse von Hashfunktionen (mit besonderen Eigenschaften) auswählen.

$\rightarrow$ Für jede Schlüsselmenge $S \subseteq U$ gutes Laufzeitverhalten im Mittel (bezogen auf Zufallsauswahl der Hashfunktion).

Zusammenfassung

- Auch die beste Hashfunktion kann Kollisionen nicht ganz vermeiden, deshalb sind Hashverfahren im *Worst Case* ineffiziente Realisierungen der Operationen search, insert, delete, im Durchschnitt sind sie jedoch weitaus effizienter als Verfahren, die auf Schlüsselvergleichen basieren.
- Die Anzahl benötigter Schritte zum Suchen, Einfügen und Entfernen hängt (im Durchschnitt) im wesentlichen vom Belegungsfaktor, d.h. dem Verhältnis von Anzahl aktueller Schlüssel zur Größe der Hashtabelle, ab.