2.4 Suchen

2.4.2 Hashing

- Suche ein bestimmtes Element in einer gegebenen Menge
  - Größe der Menge
  - Zahl der Anfragen
  - Größe des Wertebereichs
  - ...

Beispiel-Problem

- Geg. zwei Mengen
  $A = \{a_0,\ldots,a_{m-1}\}$ und $B = \{b_0,\ldots,b_{n-1}\}$

- Wertebereich $0 \leqslant a_i, b_j < q$

- Frage: $A \subseteq B$?

- Parameter: m, n, q

Naiver Algorithmus

- SubSet(A[0..m−1],B[0..n−1])

```
    for i ← 0 to m−1 do
        j ← 0
        while j < n and A[ i ] ≠ B[ j ] do
            j ← j+1
        if j = n then
            return false
    return true
```

O(n)

O(n×m)

Naiver Algorithmus

- SubSetSort(A[0..m−1],B[0..n−1])

```
sort(B)
for i ← 0 to m−1 do
    if not Find(A[ i ],B) then
        return false
return true
```

Naiver Algorithmus

- Find(a,B[0..n−1])
```
for i ← 0 to n−1 do
    if a = B[ i ] then
        return true
return false
```
- Aufwand $O(n)$

Naiver Algorithmus

- Find(a,B[0..n−1])
```
l ← 0
r ← n
while l < r do
    m ← ( l + r ) / 2
    if       a < B[m]    then  r ← m
    elseif   a > B[m]    then  l ← m + 1
    else     return true
return false
```
- Aufwand $O(\log n)$

Naiver Algorithmus

- SubSetSort(A[0..m−1],B[0..n−1])

```
    sort(B)                              O(n×logn)
    for i ← 0 to m−1 do
O(logn)  if not Find(A[ i ], B) then     O(m×logn)
            return false
    return true
```

- Aufwand $O((n+m)\times\log n)$

Kleiner Wertebereich

- Wenn der Wertebereich klein ist, können Mengen als Array of Bool ("Bitvektor") implementiert werden

- setA[0,…,q−1] $\in \{0,1\}^q$

Kleiner Wertebereich

- SubSet(A[0..m−1],B[0..n−1])

```
setA[0..q−1] ← 0
setB[0..q−1] ← 0
for i ← 0 to m−1 do
    setA[ A[ i ] ] ← 1
for i ← 0 to n−1 do
    setB[ B[ i ] ] ← 1
for i ← 0 to q−1 do
    if setA[ i ] = 1 and setB[ i ] = 0 then
        return false
return true
```

Kleiner Wertebereich

- SubSet(A[0..m−1],B[0..n−1])

```
O(q)    setA[0..q−1] ← 0
        setB[0..q−1] ← 0
O(m)    for i ← 0 to m−1 do
            setA[ A[ i ] ] ← 1
O(n)    for i ← 0 to n−1 do
            setB[ B[ i ] ] ← 1
O(q)    for i ← 0 to q−1 do
            if setA[ i ] = 1 and setB[ i ] = 0 then
                return false
        return true
```

Kleiner Wertebereich

- SubSet(A[0..m−1],B[0..n−1])

O(q)

```
setB[0..q−1] ← 0
```

O(n)

O(m)

```
for i ← 0 to n−1 do
    setB[ B[ i ] ] ← 1
for i ← 0 to m−1 do
    if setB[ A[ i ] ] = 0 then
        return false
return true
```

Kleiner Wertebereich

- Gesamtaufwand
  $O(n+m+q) = O(n+m)$

- Wertebereich wird als konstant angenommen

- Anwendung bei Mehrfachmengen

Mittlerer Wertebereich

- Implementierung des Datentyps Menge als Array of Bool ist elegant, da Einfügen, Zugriff und Löschen in O(1) realisiert werden können.

- Probleme
  - Speicherbedarf O(q)
  - Initialisierung O(q)

Lazy Initialization

- Verwende
  - Array of Bool B[0..q−1]
  - Array of Int P[0..q−1]
  - Array of Int Q[0..q−1]
  - int top

Lazy Initialization

- top: Anzahl der bereits initialisierten Einträge im Array B[]
- P[], Q[]: Zeigen an, ob der entsprechende Eintrag bereits initialisiert ist
  - P[]: in welchem Insert/Delete Schritt
  - Q[]: Bestätigung, dass P[] gültig ist

Lazy Initialization

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	?	?	?	?	?	?	?	?	?	?	?
P[]	4	7	2	9	10	2	2	4	6	3	1	8	2	7
Q[]	8	2	6	6	9	2	1	8	9	2	2	11	3	1

Lazy Initialization

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	?	?	?	?	?	?	?	?	?	?	?
P[]	4	7	2	9	10	2	2	4	6	3	1	8	2	7
Q[]	8	2	6	6	9	2	1	8	9	2	2	11	3	1

Zufallseinträge

top

Lazy Initialization

Insert(6)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	?	?	?	?	?	?	?	?	?	?	?

$2 \geqslant$ top

$\Rightarrow$ invalid

P[]	4	7	2	9	10	2	2	4	6	3	1	8	2	7

Q[]	8	2	6	6	9	2	1	8	9	2	2	11	3	1

$\uparrow$

top

Lazy Initialization

Insert(6)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	?	?	?	T	?	?	?	?	?	?	?
P[]	4	7	2	9	10	2	0	4	6	3	1	8	2	7
Q[]	6	2	6	6	9	2	1	8	9	2	2	11	3	1

top

Lazy Initialization

Insert(3)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	?	?	?	T	?	?	?	?	?	?	?
P[]	4	7	2	9	10	2	0	4	6	3	1	8	2	7
Q[]	6	2	6	6	9	2	1	8	9	2	2	11	3	1

$9 \geqslant$ top

$\Rightarrow$ invalid

top

Lazy Initialization

Insert(3)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	T	?	?	T	?	?	?	?	?	?	?
P[]	4	7	2	1	10	2	0	4	6	3	1	8	2	7
Q[]	6	3	6	6	9	2	1	8	9	2	2	11	3	1
			top ↑											

Lazy Initialization

Insert(10)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	T	?	?	T	?	?	?	?	?	?	?
P[]	4	7	2	1	10	2	0	4	6	3	1	8	2	7
Q[]	6	3	6	6	9	2	1	8	9	2	2	11	3	1

$1 <$ top but $Q[1] \neq 10$
$\Rightarrow$ invalid

↑ top

Lazy Initialization

Insert(10)

B[]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
?	?	?	T	?	?	T	?	?	?	T	?	?	?

P[]

4	7	2	1	10	2	0	4	6	3	2	8	2	7

Q[]

6	3	10	6	9	2	1	8	9	2	2	11	3	1

top

Lazy Initialization

Delete(6)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	T	?	?	T	?	?	?	T	?	?	?

0 < top and Q[0] = 6
⇒ valid

P[]	4	7	2	1	10	2	0	4	6	3	2	8	2	7
Q[]	6	3	10	6	9	2	1	8	9	2	2	11	3	1

top

Lazy Initialization

Delete(6)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	T	?	?	F	?	?	?	T	?	?	?
P[]	4	7	2	1	10	2	0	4	6	3	2	8	2	7
Q[]	6	3	10	6	9	2	1	8	9	2	2	11	3	1

top

Lazy Initialization

Find(3)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	T	?	?	F	?	?	?	T	?	?	?
P[]	4	7	2	1	10	2	0	4	6	3	2	8	2	7
Q[]	6	3	10	6	9	2	1	8	9	2	2	11	3	1

1 < top and Q[1] = 3
⇒ valid

top

Lazy Initialization

Find(3) ⇒ T

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	T	?	?	F	?	?	?	T	?	?	?

1 < top and Q[1] = 3
⇒ valid

P[]	4	7	2	1	10	2	0	4	6	3	2	8	2	7

Q[]	6	3	10	6	9	2	1	8	9	2	2	11	3	1

top

Lazy Initialization

Find(4)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	T	?	?	F	?	?	?	T	?	?	?
P[]	4	7	2	1	10	2	0	4	6	3	2	8	2	7
Q[]	6	3	10	6	9	2	1	8	9	2	2	11	3	1

$10 \geqslant$ top

$\Rightarrow$ invalid

↑ top

Lazy Initialization

Find(4) ⇒ F

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	T	?	?	F	?	?	?	T	?	?	?

$10 \geqslant$ top

⇒ invalid

P[]	4	7	2	1	10	2	0	4	6	3	2	8	2	7

Q[]	6	3	10	6	9	2	1	8	9	2	2	11	3	1

↑

top

Lazy Initialization

Find(12)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	T	?	?	F	?	?	?	T	?	?	?
P[]	4	7	2	1	10	2	0	4	6	3	2	8	2	7
Q[]	6	3	10	6	9	2	1	8	9	2	2	11	3	1

2 < top but Q[2] ≠ 12
⇒ invalid

top

Lazy Initialization

Find(12) ⇒ F

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
B[]	?	?	?	T	?	?	F	?	?	?	T	?	?	?
P[]	4	7	2	1	10	2	0	4	6	3	2	8	2	7
Q[]	6	3	10	6	9	2	1	8	9	2	2	11	3	1

2 < top but Q[2] ≠ 12
⇒ invalid

top

Lazy Initialization

- Init()

  top ← 0

- Insert(k)

  if P[k] < top and Q[P[k]] = k then

  B[k] ← true

  else

  P[k] ← top

  Q[top] ← k

  B[k] ← true

  top ← top+1

Lazy Initialization

- Delete(k)

```
if P[k] < top and Q[P[k]] = k then
    B[k] ← false
else
    P[k] ← top
    Q[top] ← k
    B[k] ← false
    top ← top+1
```

Lazy Initialization

- Find(k)

  if P[k] < top and Q[P[k]] = k then
      return B[k]
  return false

Großer Wertebereich

- z.B. Matrikelnummern: 6-stellig
- Große Wertebereiche sind in der Regel nicht dicht besetzt
- Bilde den Wertebereich W auf eine (kleinere) Indexmenge J ab

Hash-Funktion

- Hash-Funktion $F: W \rightarrow J$
- Schlüssel $F(w) = j$
- Effiziente Suche in der Index-Menge J, wenn die Funktion F einfach ist.
- In der Regel $\#J \ll \#W$
- Kollisionen: $F(w)=F(w')$ obwohl $w \neq w'$
- Wie häufig sind Kollisionen?

Hashing

- Speichere Elemente $a_1,\ldots,a_n \in W$ in einer Tabelle der Größe $O(n)$
- Hash-Funktion $F: W \to J$
  - $W \subseteq N$: beliebig großer Wertebereich
  - $J \subseteq N$: Hash-Tabelle
- Schlüssel $F(w) = j$
- Wähle F so, dass die Schlüssel $F(a_i)$ gleichverteilt in J liegen

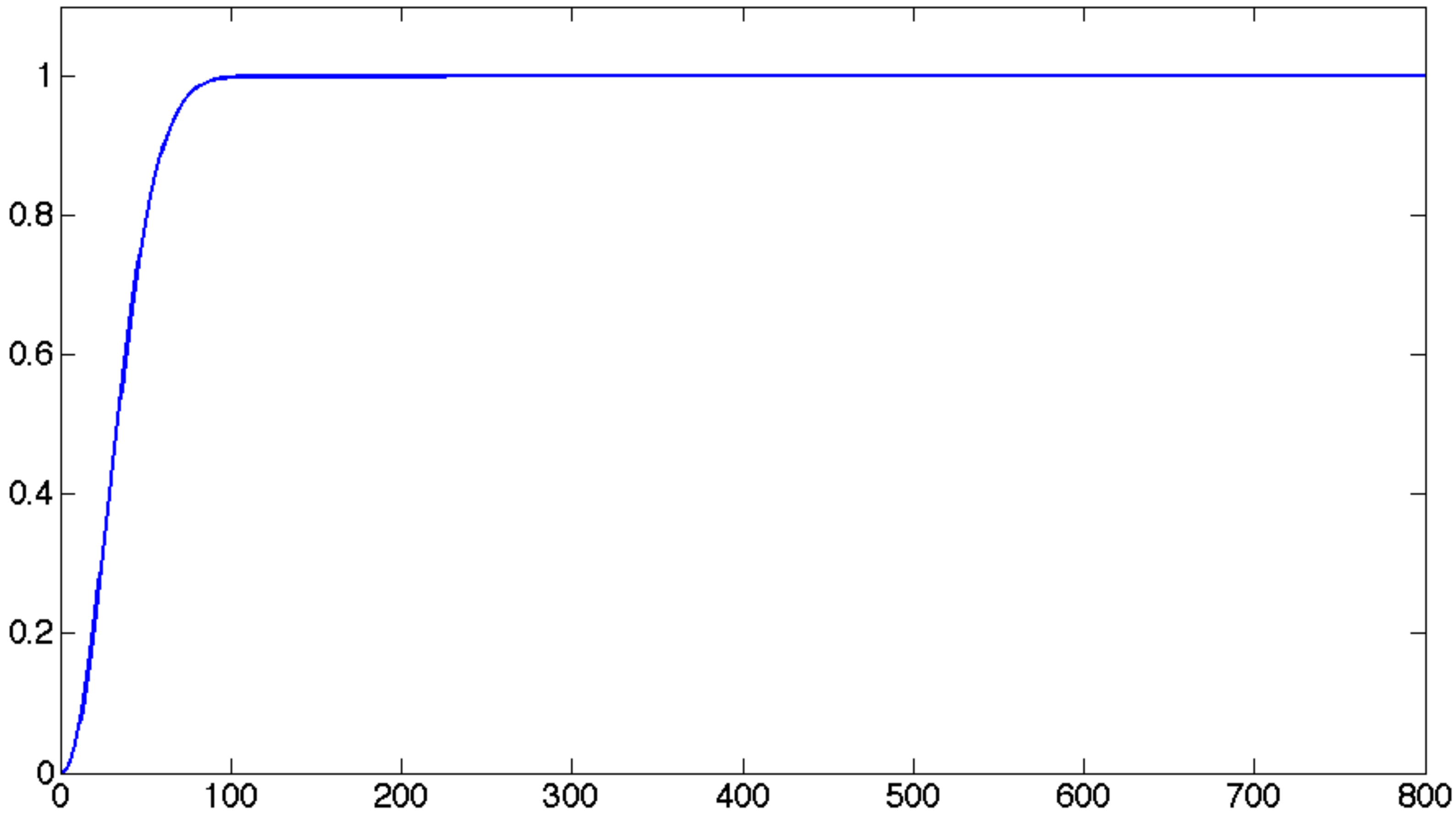
Hashing

- Bucket-Sort
  - Gleichverteilte Schlüssel
  - Berechne Speicheradresse aus Schlüssel
  - Erwarteter Füllgrad der Buckets O(1)
- Hashing
  - Berechne Index aus dem Schlüssel
  - Wähle die Größe der Indexmenge in der Größenordnung der Zahl der Schlüssel
  - Erzeuge Gleichverteilung durch “Streuung”

Kollisionen

- Identifiziere Studenten über die Matrikelnummer: 10^6 Möglichkeiten
- Studenten in dieser Vorlesung < 800
- Finde eine Funktion F, die den Bereich $[0..10^6-1]$ möglichst gleichverteilt in den Bereich $[0..799]$ abbildet (z.B. mod 800).

Kollisionen



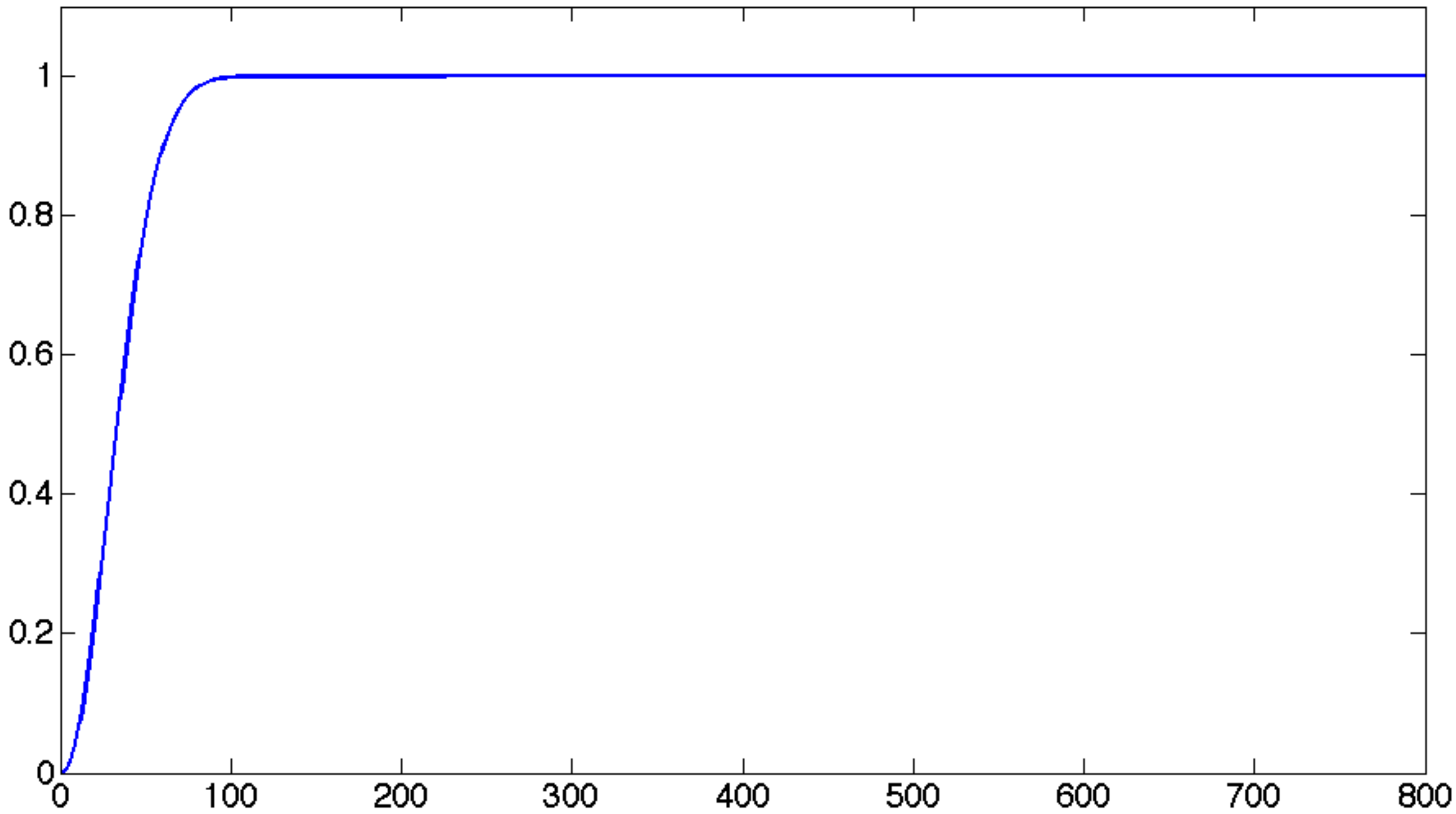
Kollisionen

- Kollisionswahrscheinlichkeit 50% schon bei 34 Studenten erreicht!
- Bei 50 Studenten bereits 79%
- vgl. Geburtstagsparadoxon

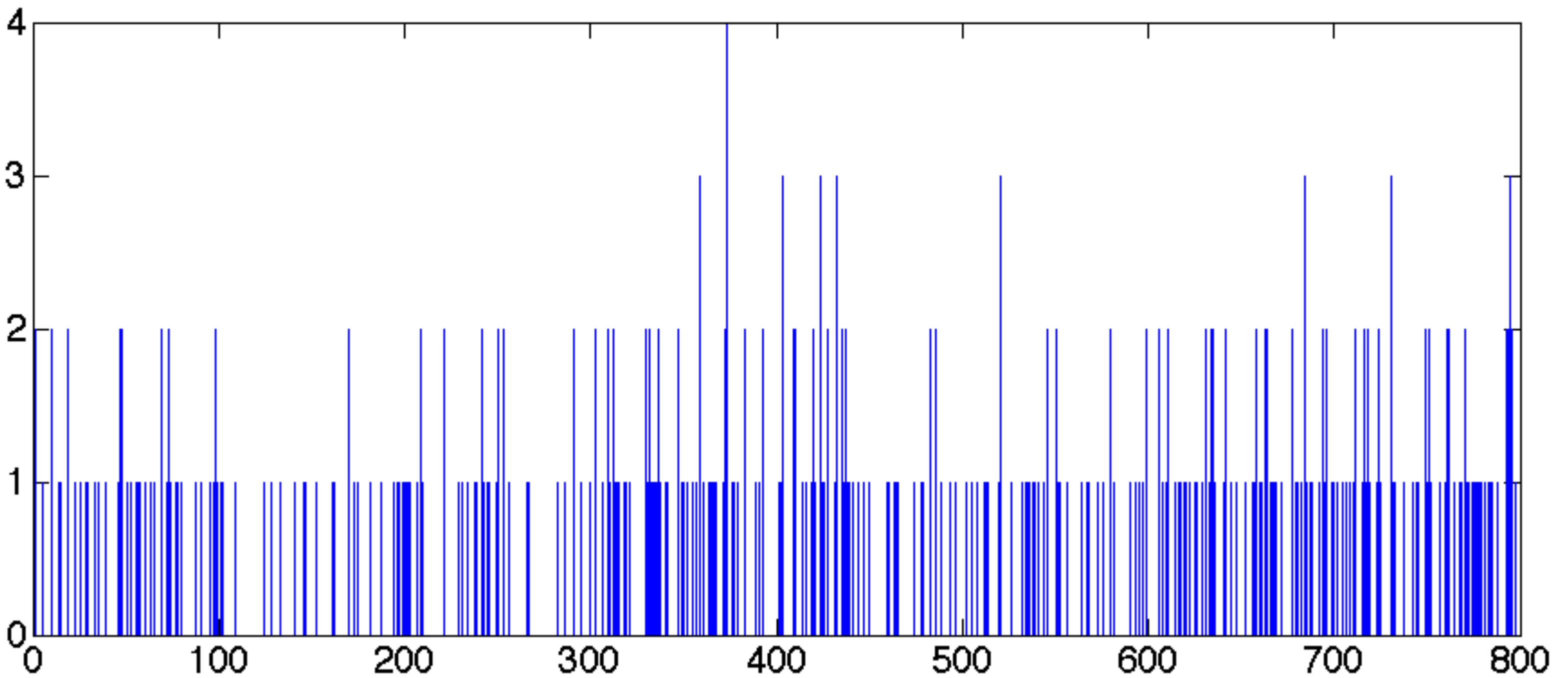
Kollisionen

- Wahrscheinlichkeit für Kollisionsfreiheit
  - 1. Student: $P_1 = 800/800$
  - 2. Student: $P_2 = 800/800 \times 799/800$
  - …
  - k. Student: $P_k = 800/800 \times \ldots \times (801-k)/800$
- Kollisionswahrscheinlichkeit: $Q_k = 1-P_k$

Kollisionen



Kollisionen



Kollisionen

- 384 Studenten sind im Übungssystem registriert ...
  - 61 einfache Kollisionen
  - 9 mehrfache Kollisionen
  - Obwohl: Tabelle nur zu 48% gefüllt

Hashing

- Finde gute Hash-Funktion
  - Surjektivität
  - Gute Streuung auch bei kohärenten Original-Schlüsseln

- Kollisionsbehandlung
  - geschlossen
  - offen

Hash-Funktionen

- Perfekte Hash-Funktion
  - $S \subseteq W = \{0,\ldots,N-1\}$, $\#W=N$, $\#S=n$
  - Finde eine Funktion F, die die Elemente von S injektiv auf die Indexmenge $\{0,\ldots,m-1\}$ abbildet $(m \geqslant n)$
  - Existiert immer, aber schwer zu finden
  - Für $m \geqslant 3 \times n$ in $O(n \times N)$

Hash-Funktionen

- Universelle Hash-Funktion
  - Erwarteter Aufwand für Elementzugriff $O(1)$
  - Worst-case Aufwand $O(n)$ (vgl. Bucket-Sort)
  - Hängt von der Struktur der Menge $S \subseteq W$ ab
  - Definiere eine Menge $H=\{F_i()\}$ von Hashfunktionen, so dass bei $w \neq w'$ $\#\{\ F \in H:\ F(w)=F(w')\ \} \leqslant \#H/m$ ist (m = Größe der Indexmenge)
  - Wähle Hash-Funktion aus dieser Menge zufällig aus → gutes Durchschnittsverhalten bei Kollisionen

Hash-Funktionen

- Präfix, Suffix, …
- $F(w) = w \bmod q$
  - $q = 2^k \rightarrow$ benutzt nur die letzten k Binärstellen
  - $q = 10^k \rightarrow$ benutzt nur die letzten k Dezimalstellen
  - q prim $\rightarrow$ optimale Streuung
- $F(w) = (w \times s + t) \bmod q$
  - Beliebige q möglich, besser: Primzahlen

Hash-Funktionen

- Problem: Tabellengröße ≠ Primzahl
- $G(w) = (w \times s + t) \bmod q$
  (q große Primzahl)
- $F(w) = G(w) \bmod q'$
  (q' Tabellengröße)

- Mittel-Quadrat-Methode
  - Seien die Originalschlüssel p-stellige Dezimalzahlen
  - Quadrate sind $2p$-stellig
  - Tabellengröße 10^q, $q < p$
  - $F(w) = \lfloor 10^{-\lceil p/2 \rceil} w^2 \rfloor \bmod 10^q$
  - Hängt von allen p Stellen ab

Kollisionsbehandlung

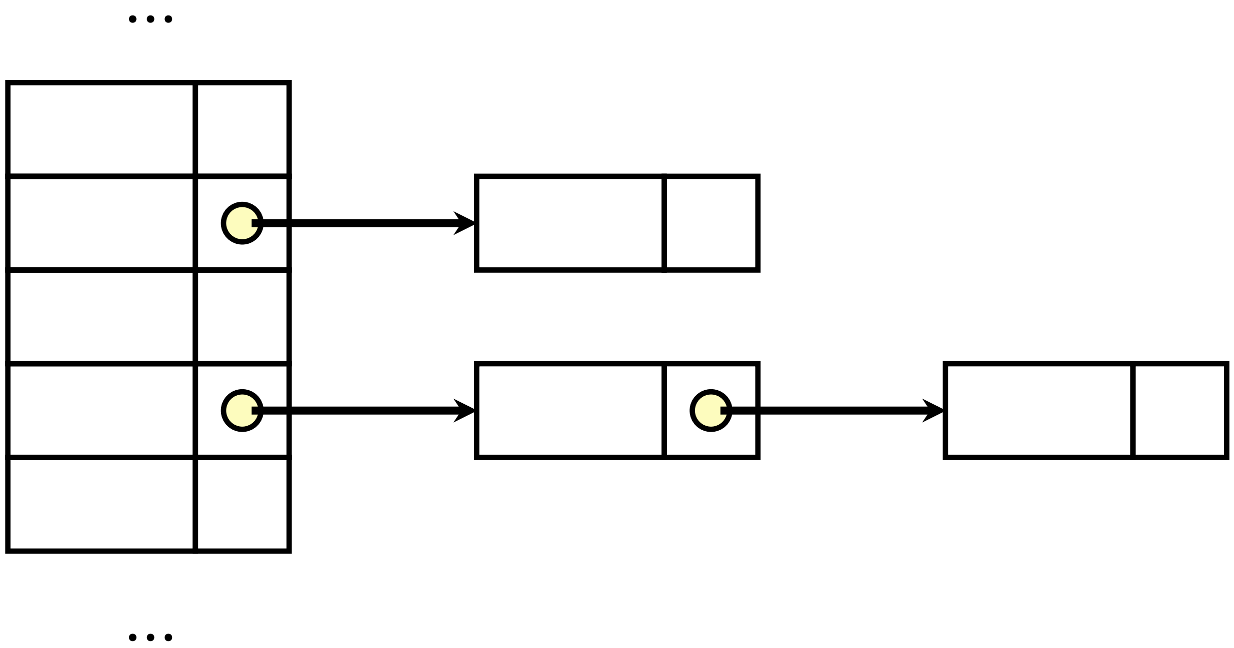
- Offenes Hashing
(geschlossene Adressierung)

- Geschlossenes Hashing
(offene Adressierung)

Offenes Hashing

- Jeder Tabelleneintrag ist Anchor-Element einer verketteten Liste
- Alle Objekte mit gleichem Hash-Key werden in die entsprechende Liste eingefügt
- Nachteil: es wird dynamisch mehr Speicherplatz belegt
- Vorteil: es werden keine alternativen Adressen berechnet

Offenes Hashing



Offenes Hashing

- Erwartete Länge der Listen bei Belegungs-faktor $a = n/m$ ist ebenfalls gleich a (n = Zahl der Objekte, m = Größe der Tabelle)
- Vgl. Analyse von Bucket-Sort
- Erwarteter Aufwand: $O(1+a)$
  - $a \ll 1 \to$ Array-Verhalten
  - $a = 1$
  - $a \gg 1 \to$ Listen-Verhalten

Geschlossenes Hashing

- Wenn eine Kollision auftritt (weil der adressierte Tabellenplatz schon belegt ist) wird ein alternativer Index berechnet
- Vorteil: es wird kein neuer Speicherplatz belegt
- Nachteil: es werden andere Adressen verwendet

Geschlossenes Hashing

- Anstatt nur den Index F(w) zu verwenden, wird eine Sequenz von Indizes

  F(w,0), F(w,1),…,F(w,m-1) berechnet.
- Das Objekt w wird im ersten freien Tabellenplatz dieser Sequenz gespeichert
- Nachteil: Die Sequenz muss bei jedem Zugriff durchsucht werden

Geschlossenes Hashing

- Nachteil: Die Sequenz muss bei jedem Zugriff durchsucht werden
  - Positiver Fall → erwartet bis zur Hälfte
  - Negativer Fall → immer bis zum Ende

- Die Sequenzen sind typischerweise länger als die Listen beim offenen Hashing!

Kollisionsauflösung

- Lineares Sondieren
  $F(w,i) = (F(w) + i) \bmod m$

- Führt in der Regel zu Cluster-Bildung, da jedes Element, das mit einem kleinen Cluster kollidiert, an dessen Ende eingefügt wird und damit den Cluster vergrößert

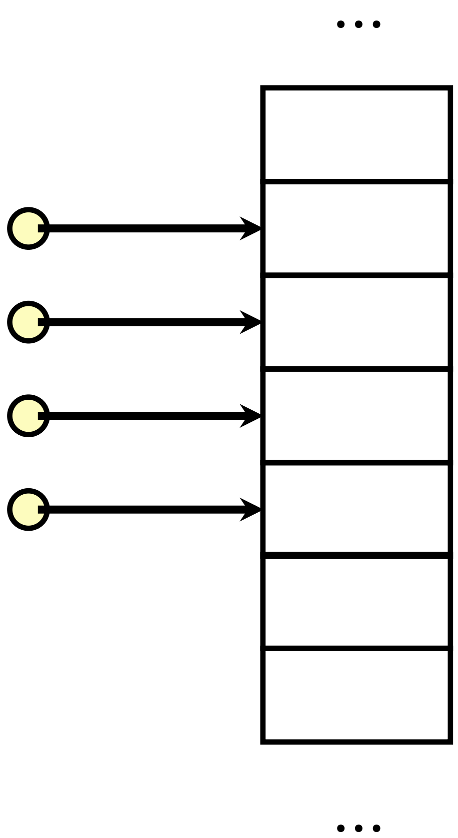
Lineares Sondieren

F(w) = F(w,0)

F(w,1)

F(w,2)

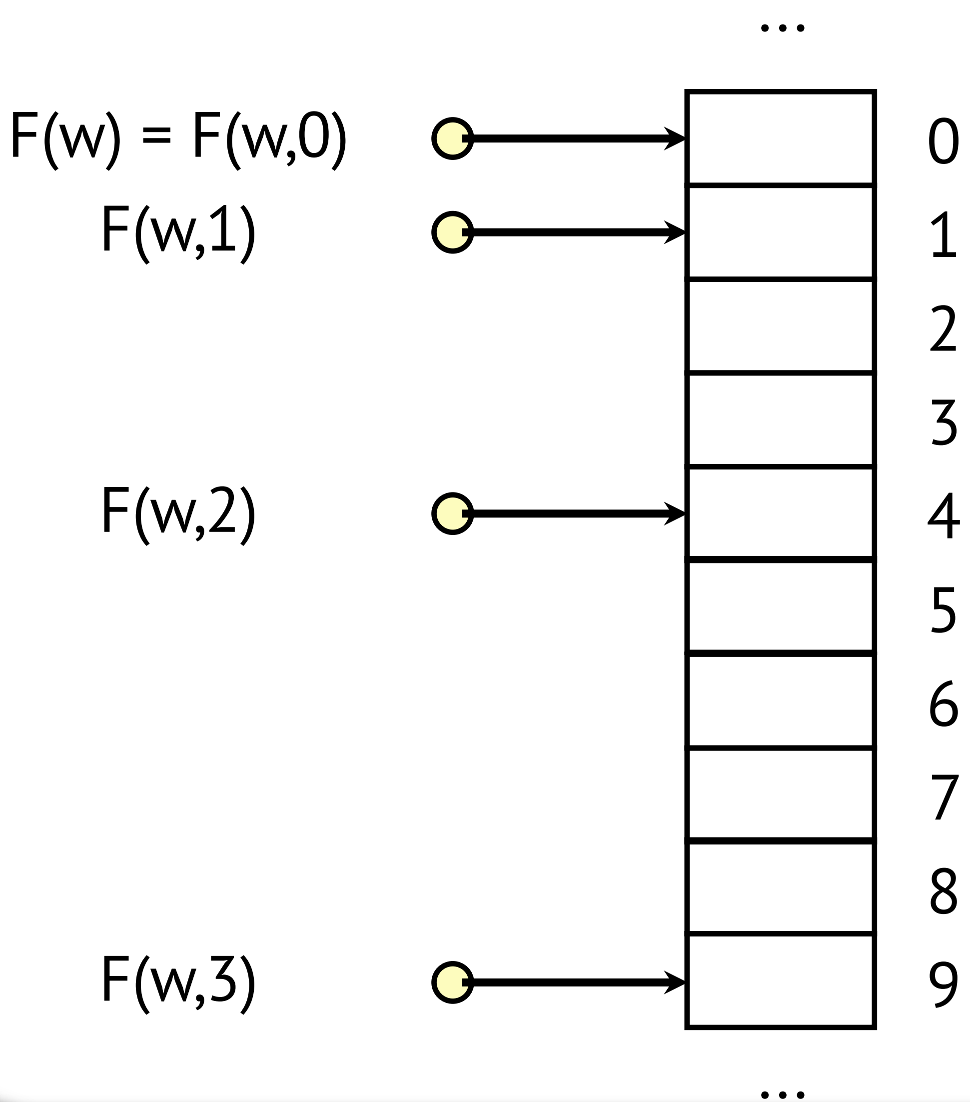
F(w,3)



Kollisionsauflösung

- Quadratisches Sondieren
  $F(w,i) = (F(w) + i^2) \bmod m$
- Besseres Streuverhalten, insbesondere wenn m eine Primzahl ist.
- Noch besser: m Primzahl mit $m = 4 \times j + 3$
  $F(w, 2 \times i+1) = (F(w) + i^2) \bmod m$
  $F(w, 2 \times i) = (F(w) - i^2) \bmod m$

Quadratisches Sondieren



Kollisionsauflösung

- Doppeltes Hashing

  $F(w,i) = (F_1(w) + F_2(w) * i) \bmod m$

  wobei m teilerfremd zu allen Werten $F_2(w)$

- Fast ideales Verhalten, Kollisionswahrscheinlichkeiten

  $P(\, F_1(w) = F_1(w') \bmod m \,) = 1/m$

  $P(\, F_2(w) = F_2(w') \bmod m \,) = 1/m$

  $P(\, F_1(w) = F_1(w') \bmod m \ \wedge$

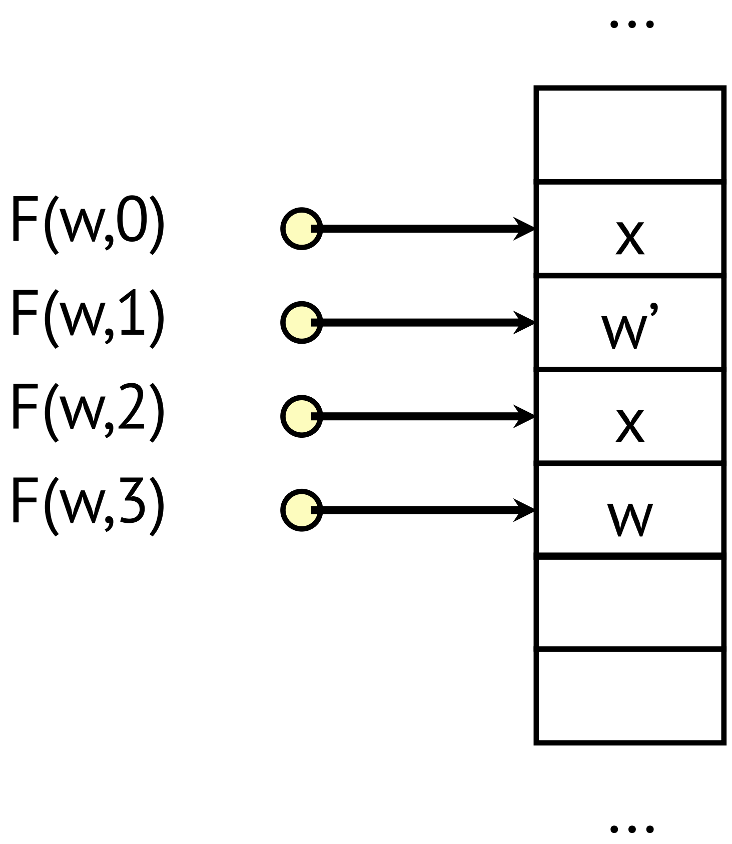
  $F_1(w)+F_2(w) * i =$

  $F_1(w')+F_2(w') * i \bmod m \,) = 1/m^2$

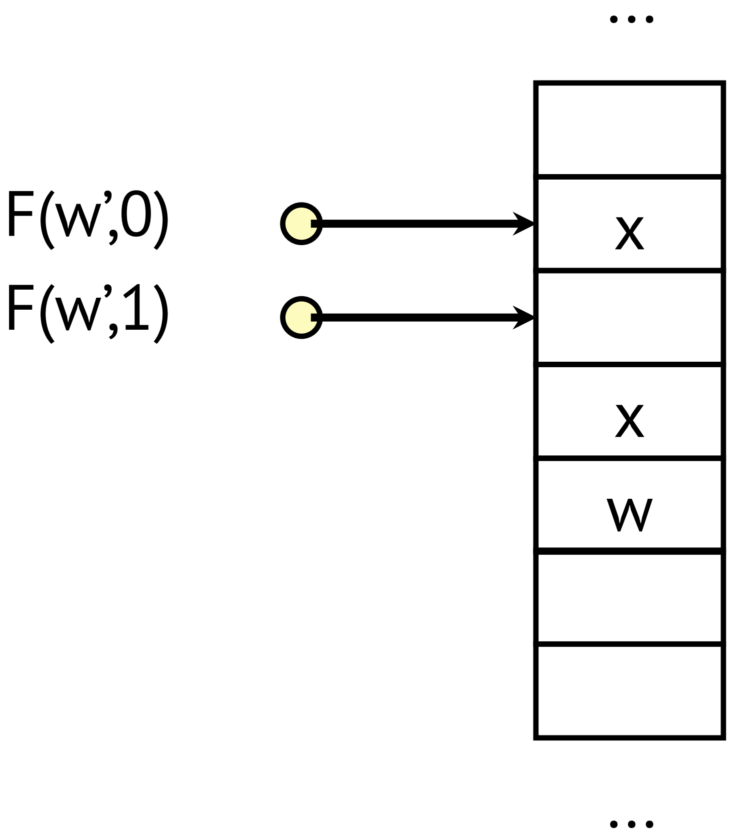
Löschen

- Bei offenem Hashing einfach
- Bei geschlossenem Hashing
  - Markiere Tabelleneinträge als gelöscht
  - Gelöschte Einträge können überschrieben werden
  - Gelöschte Einträge werden beim Suchen berücksichtigt

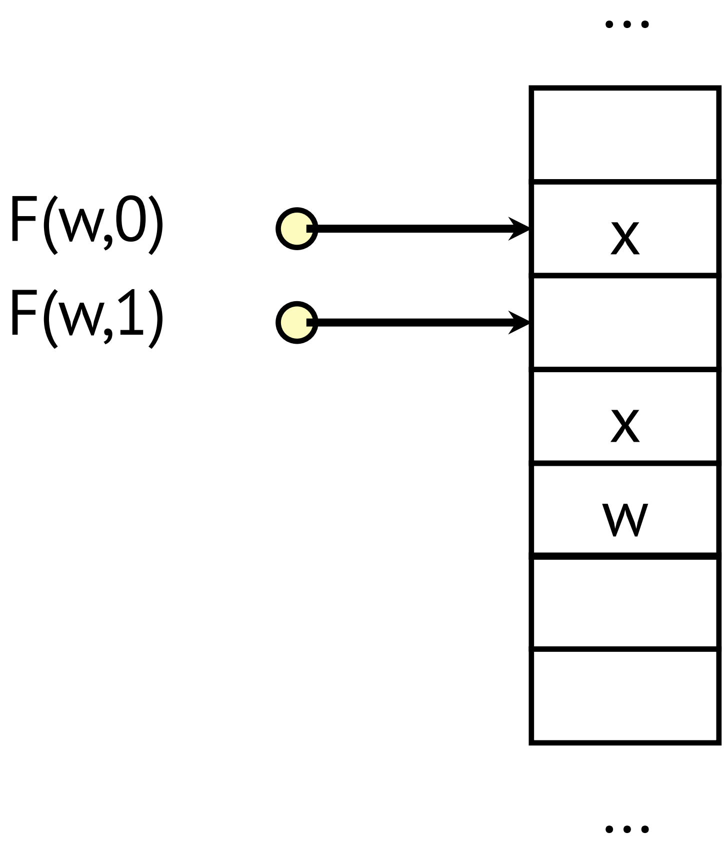
Löschen



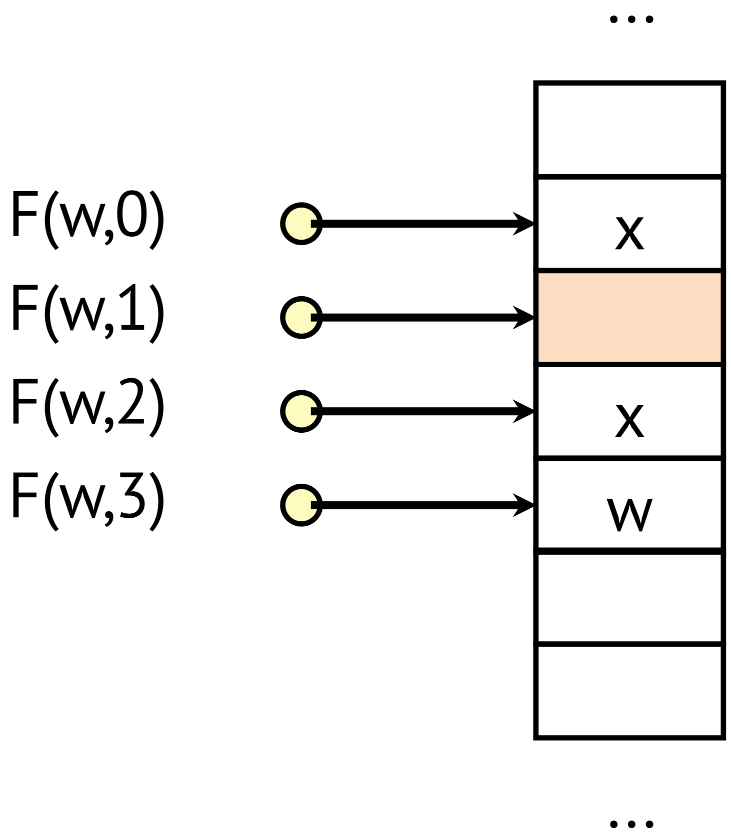
Löschen



Löschen



Löschen



Aufwandsabschätzung

- Wie viele Kollisionen treten auf?
- Wie oft muss sondiert werden?
- Annahme: Ideale Hash-Funktion (aber nicht perfekt)
- Schlüssel werden gleichverteilt

Aufwandsabschätzung

- Q(i,n,m) Wahrscheinlichkeit, dass
  - mindestens i Sondierungsschritte
  - bei bereits n Elementen
  - in einer Tabelle der Größe m

  notwendig sind

Aufwandsabschätzung

- $Q(0,n,m) = 1$
- $Q(1,n,m) = n/m$
- $Q(2,n,m) = n/m \times (n-1)/(m-1)$
- …
- $Q(i,n,m) = n/m \times \ldots \times (n+1-i)/(m+1-i)$

Aufwandsabschätzung

- Mittlere Kosten für das Einfügen des (n+1)sten Elements:

$$
\begin{aligned}
C_{ins}(n,m) &= \sum_{j=0}^{n} (j+1)(Q(j,n,m) - Q(j+1,n,m)) \\
&= \sum_{j=0}^{n} (j+1)Q(j,n,m) - \sum_{j=1}^{n+1} jQ(j,n,m) \\
&= \sum_{j=0}^{n} Q(j,n,m) - (n+1)Q(n+1,n,m)
\end{aligned}
$$

Aufwandsabschätzung

- Mittlere Kosten für das Einfügen des (n+1)sten Elements:

$$\begin{aligned} C_{ins}(n,m) &= \sum_{j=0}^{n} Q(j,n,m) - (n+1)Q(n+1,n,m) \\ &= \sum_{j=0}^{n} Q(j,n,m) \\ &= \frac{m+1}{m+1-n} \end{aligned}$$

Aufwandsabschätzung

- Mittlere Kosten für das Einfügen des (n+1)sten Elements:

$$C_{ins}(n,m) \approx \frac{m+1}{m+1-n}$$

- Sei $a = n/m$, dann gilt

$$C_{ins}(n,m) \approx \frac{1}{1-a}$$

Aufwandsabschätzung

$$C_{ins}(n,m) \approx \frac{1}{1-a}$$

a	50%	70%	90%	95%	99%	99.9%
C_{ins}	2	3.3	10	20	100	1000

Aufwandsabschätzung

- Suchen eines Elements in der Hash-Tabelle
  - Es muss dieselbe Sondierungssequenz zur Kollisionsauflösung abgearbeitet werden
  - Negative Suche endet beim ersten freien Tabelleneintrag
    $\rightarrow C_{search}(n,m) = C_{ins}(n,m)$
  - Positive Suche bricht früher ab...

Aufwandsabschätzung

- Positive Suche
  - Kosten entsprechen den Kosten beim Einfügen (gleiche Sondierungssequenz)
  - Bei welchem Füllgrad wurde das Element eingefügt?
  - Bilde Mittelwert über alle möglichen Einfüge-Reihenfolgen

Aufwandsabschätzung

$$C_{search}(n,m) = \frac{1}{n}\sum_{j=0}^{n-1} C_{ins}(j,m) = \cdots = \frac{-ln(1-a)}{a}$$

a	50%	70%	90%	95%	99%	99.9%
C_{search}	1.38	1.72	2.56	3.15	4.65	6.91