## 5.4 Analyse der Hashverfahren

Beim linearen Sondieren steigt die Zeit, die man für das Einfügen benötigt, wegen der Cluster mit steigendem Grad der Auslastung (d.h., wenn sich die Anzahl k der eingetragenen Schlüssel der Zahl p der Plätze in der Tabelle nähert) überproportional an. Beim quadratischen Sondieren tritt dies nicht so stark hervor. Beim Doppel-Hash-Verfahren noch weniger. (Dafür wird das Löschen jedes Mal schwieriger.)

Welche theoretischen Ergebnisse gibt es zur Analyse der Laufzeiten beim Suchen und Einfügen?



Für die Beweise benötigt man einige Annahmen. Diese fordern meist die Gleichverteilung der Schlüssel und die Unabhängigkeit von Ereignissen.

Annahmen:

1. Die Hashfunktion f ist gleichverteilt über die Schlüsselmenge, sie bevorzugt oder benachteiligt dort keine Bereiche.
2. Jeder Schlüssel ist bei beim Suchen und beim Einfügen gleichwahrscheinlich.
3. Erfolgt beim Einfügen eines Schlüssels eine Kollision, so werden bis zu dessen Eintrag auf einen freien Platz nur paarweise verschiedene Plätze besucht.

Wie lange dauert es unter diesen Annahmen im Mittel, einen Schlüssel in eine Hashtabelle einzufügen, in der bereits k von p Plätzen belegt sind?

Setze

$\mathbf{w_i}$ = Wahrscheinlichkeit dafür, dass für dieses Einfügen genau i Vergleiche durchgeführt werden ($1 \le i \le k+1$).

Wegen der Annahmen gilt: Mit der Wahrscheinlichkeit k/p trifft man beim ersten Mal auf einen belegten Platz, mit der Wahrscheinlichkeit (k-1)/(p-1) beim zweiten Mal, mit (k-2/(p-2) beim dritten Mal usw. So erhalten wir die Formeln:



$w_1 = 1 - k/p$

$w_2 = (k/p)\cdot(1 - (k-1)/(p-1))$ , allgemein:

$w_i = (k/p)\cdot(k-1)/(p-1)\cdot(k-2)/(p-2)\cdot\ldots\cdot(k-i+2)/(p-i+2)\cdot(1 - (k-i+1)/(p-i+1))$

$$= \frac{k\cdot(k-1)\cdot(k-2)\cdot\ldots\cdot(k-i+2)}{p\cdot(p-1)\cdot(p-2)\cdot\ldots\cdot(p-i+2)} \left(1 - \frac{k-i+1}{p-i+1}\right)$$

Dann lautet die mittlere Zahl der Vergleiche $\mathbf{E_{k+1}}$ beim Einfügen eines (k+1)-ten Schlüssels in eine Hashtabelle der Größe p:

$$\mathbf{E_{k+1}} = \sum_{i=1}^{k+1} i\cdot w_i = \ldots = \frac{p+1}{p+1-k} = \frac{1}{1-\lambda} \qquad \text{mit } \lambda = k/(p+1)$$

$\lambda \approx$"Auslastungsgrad" k/p

*(Dieses Ergebnis wird an der Tafel in der Vorlesung ausgerechnet. Versuchen Sie es auch selbst einmal.)*

**Satz:** (Beachte die Annahmen drei Folien zuvor.)

Um den (k+1)-ten Schlüssel in eine Hashtabelle der Größe p einzufügen, werden im Mittel $\frac{p+1}{p+1-k} = \frac{1}{1-\lambda}$ Vergleiche (mit $\lambda = k/(p+1)$) benötigt.

Einige Funktionswerte für $E_{k+1} = \frac{p+1}{p+1-k}$

| $\lambda$ | $E_{k+1}$ | $\lambda$ | $E_{k+1}$ | $\lambda$ | $E_{k+1}$ |
|---|---|---|---|---|---|
| 0,1 | 1,11 | 0,5 | 2,00 | 0,85 | 6,67 |
| 0,2 | 1,25 | 0,6 | 2,50 | 0,88 | 8,33 |
| 0,3 | 1,43 | 0,7 | 3,33 | 0,90 | 10,00 |
| 0,4 | 1,67 | 0,8 | 5,00 | 0,95 | 20,00 |



Wie lange dauert bei "idealen Hashfunktionen" die erfolgreiche Suche im Mittel und wie lange die nicht erfolgreiche Suche?

Die nicht-erfolgreiche Suche entspricht dem Einfügen eines neuen Schlüssels; sie wird also durch $E_{k+1}$ beschrieben.

Es sei $S_k$ die Zahl der Vergleiche, die benötigt werden, um einen Schlüssel zu finden, der in einer Hashtabelle der Größe p mit dem Auslastungsgrad k/p steht.

Die erfolgreiche Suche kann man dann durch folgende Formel beschreiben:

$$S_k = \frac{1}{k} \sum_{i=0}^{k-1} E_{i+1} \ ,$$

denn der gesuchte Schlüssel muss in einem der Schritte 1, 2, 3, ..., k in die Tabelle eingefügt worden sein, und nach der Annahme 2 können wir den Mittelwert der Zahl der Vergleiche nehmen. Durch Auswerten dieser Formel erhält man:

$$S_k \approx \frac{1}{\lambda} \cdot \ln\left(\frac{1}{1-\lambda}\right)$$

*(Dieses Ergebnis wird in den Übungen ausgerechnet.)*



**Satz:** (Beachte die Annahmen sechs Folien zuvor.)

Für die erfolgreiche Suche nach einem Schlüssel in einer Hashtabelle der Größe p werden im Mittel $S_k \approx \frac{1}{\lambda} \cdot \ln\left(\frac{1}{1-\lambda}\right)$ Vergleiche (mit $\lambda = k/(p+1)$) benötigt.

---

Man beachte, dass $\frac{1}{\lambda} \cdot \ln\left(\frac{1}{1-\lambda}\right) \leq \frac{1}{1-\lambda}$ ist wegen

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \geq 1 + x \quad \text{mit} \quad x = \frac{\lambda}{1-\lambda} \quad \text{für} \quad \lambda < 1.$$

Experimente haben ergeben, dass Doppel-Hash-Verfahren recht gut den Wert $S_k$ annähern. Wie wirken sich Kollisionen aus?

Es sei $Slin_k$ die mittlere Suchzeit für die erfolgreiche Suche, dann kann man mit einigem Aufwand beweisen (ohne Beweis hier):

$$Slin_k \approx \frac{1 - \frac{\lambda}{2}}{1 - \lambda}$$

Einige Werte zu $S_k$ und $Slin_k$ mit $\lambda = k/(p+1)$:

| $\lambda$ | $S_k$ | $Slin_k$ |
|---|---|---|
| 0,50 | 1,39 | 1,50 |
| 0,75 | 1,85 | 2,50 |
| 0,80 | 2,01 | 3,00 |
| 0,90 | 2,56 | 5,50 |
| 0,95 | 3,15 | 10,50 |
| 0,99 | 4,65 | 50,50 |

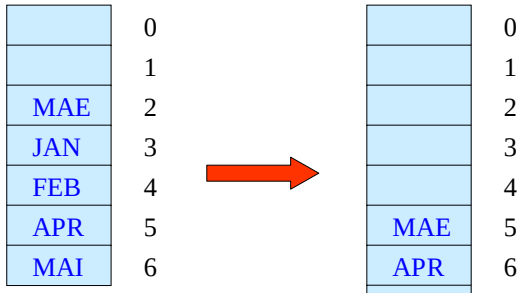
Für die Praxis, die meist mit linearem Sondieren arbeitet, folgt hieraus: Man begrenze den Auslastungs-grad möglichst auf 80%.



## 5.5 Rehashing

Was muss man tun, wenn der Auslastungsgrad über 80% hinausgeht oder gar den Wert 1 erreicht? Man muss die Hashtabelle verlängern (also p durch eine Zahl p' > p ersetzen), die neue Hashfunktion festlegen und dann eine Umorganisation der neuen Hashtabelle, in der die bisherigen Schlüssel in den Plätzen von 0 bis p-1 stehen, vornehmen.

Diesen Vorgang der Umorganisation innerhalb der bestehenden Hashtabelle bezeichnen wir als "Rehashing". Dieses Verfahren wird auch verwendet, wenn man Schlüssel, statt sie zu löschen, nur als "gelöscht" markiert, wodurch im Laufe der Zeit der Auslastungsgrad zu groß wird und eine Umorganisation mit dem gleichen p notwendig wird.




p=7

Wörter:
JAN, FEB, MAE, APR, MAI.

Hashfunktion: $f(\alpha_1\alpha_2 ...\alpha_r) = (\varphi(\alpha_1) + 2\,\varphi(\alpha_3))$ mod 7.
Lineares Sondieren.




p'=13

Alle Wörter müssen übertragen werden.

Neue Hashfunktion:
$f'(\alpha_1\alpha_2 ...\alpha_r) = (\varphi(\alpha_1)+\varphi(\alpha_3))$ mod 13.
Lineares Sondieren.

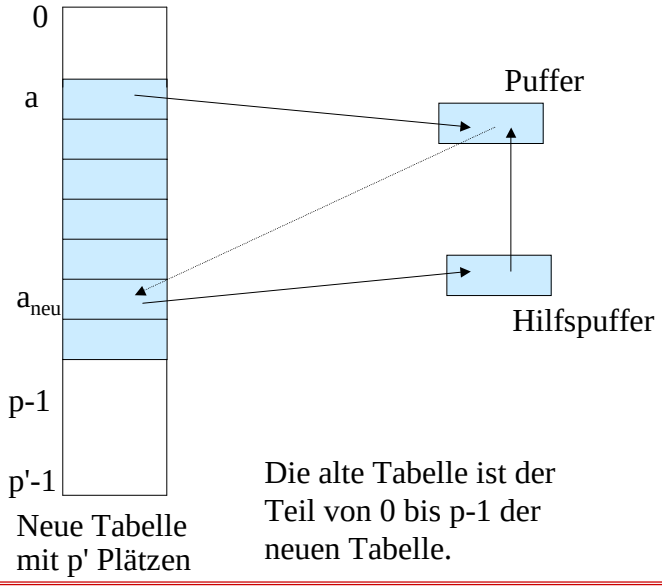
Von oben nach unten durchgehen und dabei umsortieren! Wir fangen also mit MAE, dann JAN usw.



In diesem Beispiel haben wir die Wörter JAN, FEB, MAE, APR, MAI in dieser Reihenfolge in die neue Tabelle mit p'=13 eingetragen. Dies entspricht aber nicht dem gewünschten "Rehashing", weil beispielsweise f'(MAE) = 5 ist, aber auf Platz 5 steht APR und dieses Wort würde hierbei überschrieben werden. Faktisch haben wir also nicht *innerhalb* der neuen Tabelle umorganisiert, sondern wir haben neben die alte Tabelle mit p=7 Plätzen eine neue Tabelle mit p'=13 Plätzen gelegt und die Wörter dorthin entsprechend der neuen Hashfunktion f' umgespeichert. Wir brauchten also insgesamt p+p'=20 Plätze.

Unser Rehash-Verfahren soll jedoch auf der verlängerten Ausgangstabelle arbeiten, also mit insgesamt p' Plätzen auskommen.

Rehashing: Durchlaufe die neue Tabelle von 0 bis p-1:






Erinnerung: (vgl. Anfang von Abschnitt 5.3)

```
type Eintragtyp is record
        belegt: Boolean;  geloescht: Boolean;
        kollision: Boolean;  behandelt: Boolean;
        Schluessel: Schluesseltyp;
        Inhalt: Inhalttyp;
end record;

type hashtabelle is array(0..p-1) of Eintragtyp;

A: hashtabelle;
```

## Erinnerung: (vgl. Anfang von Abschnitt 5.3)

Ganz zu Anfang wurde gesetzt:
for j in 0..p-1 loop A(j).belegt:=false; A(j).kollision:=false;
A(j).geloescht:=false; A(j).behandelt:=false; end loop;

Während des Aufbaus von Tabelle A wurden A(j).besetzt, A(j).kollision und A(j).geloescht eventuell verändert.

## Rehashing:

Puffer, Hilfspuffer: Eintragtyp;

Berechne p' als neue Größe von A. Die Hashtabelle A möge nun die Grenzen von 0 bis p'-1 besitzen.



Vorgehensweise: Führe (1) bis (3) von a=0 bis a=p-1 durch.

(1) *A(a).belegt and not A(a).gelöscht and not A(a).behandelt:*
kopiere A(a) in den Puffer und setze A(a).belegt auf false.

(2) Berechne mit der neuen Hashfunktion f' den neuen Index $a_{neu}$, wohin das Element des Puffers hingehört.

(3) Unterscheide folgende Fälle:
*not* $A(a_{neu})$*.belegt or* $A(a_{neu})$*.geloescht:* Kopiere den Puffer nach $A(a_{neu})$; setze $A(a_{neu})$.belegt und $A(a_{neu})$.behandelt auf true. Erhöhe a. Weiter bei (1).
*not* $A(a_{neu})$*.behandelt and* $A(a_{neu})$*.belegt:* Kopiere $A(a_{neu})$ in den Hilfspuffer; kopiere den Puffer nach $A(a_{neu})$; setze $A(a_{neu})$.behandelt und $A(a_{neu})$.belegt auf true. Kopiere dann den Hilfspuffer in den Puffer. Weiter bei (2).
Sonst, d.h.: $A(a_{neu})$*.behandelt:* Setze $A(a_{neu})$.kollision := true, berechne den Index $a_{neu}$ neu, weiter bei (3).

## Programmstück zum Rehashing

```
-- a durchläuft die Adressen von 0 bis p-1
-- i zählt die Zahl der auftretenden neuen Kollisionen
-- aneu gibt die Adresse an, wohin der Eintrag in der neuen
--     Tabelle gehört

for a in 0..p-1 loop
if A(a).geloescht then "lösche den Eintrag A(a)"
elsif  A(a).belegt and not A(a).behandelt  then
      Puffer := A(a);  Puffer.belegt := true;
      A(a).belegt := false;
      i := 0;

-- nun f' auf Puffer anwenden und Adresse aneu berechnen,
-- auf Kollisionen mit schon behandelten Einträgen achten
```



```
      while Puffer.belegt loop
            aneu := f'(Puffer.schluessel, i);   i := i+1;
            if  not A(aneu).belegt or A(aneu).geloescht  then
                  A(aneu) := Puffer;
                  A(aneu).geloescht := false; A(aneu).belegt := true;
                  A(aneu).behandelt := true; A(aneu).kollision:=false;
                  Puffer.belegt:= false;
            elsif  not A(aneu).behandelt  then
                  Hilfspuffer := A(aneu); Hilfspuffer.belegt := true;
                  A(aneu) := Puffer; A(aneu).behandelt := true;
                  Puffer := Hilfspuffer;
                  i := 0;
            else  A(aneu).kollision := true;    end if;
            end loop;
end if; end loop a;
```