Teil 3 der Grundvorlesung

3. Grundlegende Verfahren

~~3.1 Speicherverwaltung~~

~~3.2 Suchverfahren~~

3.3 Hashing

3.4 Sortieren

3.5 Algorithmen auf Graphen

3.6 Zufallszahlen

Gliederung des Kapitels

3.3 Hashing

3.3.1 Beispiel "modulo p"

3.3.2 Hashfunktionen

3.3.3 Offenes Hashing

3.3.4 Analyse von Hashverfahren

3.3.5 Rehashing

3.3 Hashing (gestreute Speicherung)

Grundidee:

Gegeben sei eine Menge B und eine Zahl p << |B|.

Finde eine Abbildung f: B $\rightarrow$ {0, 1, ..., p-1}, sodass es in einer zufällig ausgewählten Teilmenge A = {a_1, ..., a_n} $\subseteq$ B im Mittel nur wenige Elemente $a_i \neq a_j$ gibt mit $f(a_i) = f(a_j)$. Realisiere A in einer geeigneten Datenstruktur, mit der die folgenden drei Operationen sehr "effizient" durchgeführt werden können:

- Entscheide, ob b in A liegt (und gib ggf. an, wo). FIND
- Füge b in A ein. INSERT
- Entferne b aus A. DELETE

Zusatz: Schön wäre es, wenn auch die folgenden Operationen leicht ausführbar wären.

- Gib die Elemente von A_1 geordnet aus. SORT
- Vereinige A_1 und A_2. UNION
- Bilde den Durchschnitt von A_1 und A_2. INTERSECTION
- Entscheide, ob A_1 leer ist. EMPTINESS
- Entscheide, ob $A_1 = A_2$ ist. EQUALITY
- Entscheide, ob $A_1 \subseteq A_2$ ist. SUBSET

Die Abbildung f: B $\rightarrow$ {0, 1, ..., p-1} sollte surjektiv und gleichverteilt sein, d.h., für jedes $0 \leq m < p$ sollte die Menge $B_m = \{b \in B \mid f(b) = m\}$ ungefähr |B|/p Elemente enthalten. Weiterhin muss f schnell berechnet werden können.

Solch eine Abbildung f heißt Schlüsseltransformation oder **Hashfunktion**.

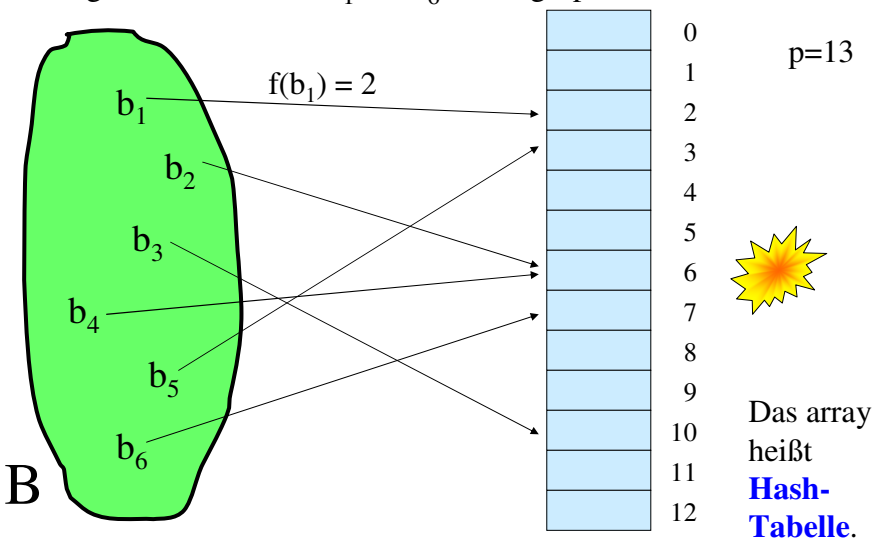
Nehmen wir an, wir hätten eine solche Abbildung
f: B $\rightarrow$ {0, 1, ..., p-1}, dann würden wir zur Speicherung von Teilmengen von B ein Feld deklarieren:
A: array (0..p-1) of <Datentyp für die Menge B>

Jedes Element b $\in$ B speichern wir unter der Adresse f(b): A(f(b)) := b.

Um festzustellen, ob ein Element b in der jeweiligen Teilmenge liegt, braucht man nur zu prüfen, was in A(f(b)) steht. Doch es entstehen Probleme, wenn in der Teilmenge mehrere Elemente mit gleichem f-Wert enthalten sind.

Wie sieht es mit den Operationen INSERT und DELETE aus? Wir schauen uns zunächst eine Skizze und dann ein Beispiel an.

Folgende 6 Elemente b_1 bis b_6 sollen gespeichert werden:



Hier ist $f(b_2) = f(b_4) = 6$. Was nun?

3.3.1 Beispiel "modulo p"

$B = \Sigma^*$ = die Menge aller Folgen über einem s-elementigen Alphabet $\Sigma = \{\alpha_0, \alpha_1, ..., \alpha_{s-1}\}$.

Weiterhin sei p eine natürliche Zahl, p>1.

Eine nahe liegende Codierung $\varphi: \Sigma \rightarrow \{0, 1, ..., s-1\}$ ist $\varphi(\alpha_i) = i$. Als Abbildung $f: \Sigma^* \rightarrow \{0, 1, ..., p-1\}$ kann man dann die Codierung eines Anfangsworts wählen (für ein q mit $0 < q \leq r$) oder Teile davon:

$$f(\alpha_{i_1}\alpha_{i_2} ...\alpha_{i_r}) = \left(\sum_{j=1}^{q} \varphi(\alpha_{i_j}) \right) \underline{\text{mod}}\ p.$$

Wir demonstrieren dies am lateinischen Alphabet, wobei wir nur die großen Buchstaben A, B, C, ... verwenden. Als Codierung φ wählen wir die Position des Buchstabens im Alphabet, wobei wir Z auf 0 abbilden (s=26):

a	φ(a)
A	1
B	2
C	3
D	4
E	5
F	6
G	7
H	8
I	9

a	φ(a)
J	10
K	11
L	12
M	13
N	14
O	15
P	16
Q	17
R	18

a	φ(a)
S	19
T	20
U	21
V	22
W	23
X	24
Y	25
Z	0

Abzubildende Menge **A** :

A = {JANUAR, FEBRUAR, MAERZ, APRIL, MAI, JUNI, JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, DEZEMBER}.

Wir erhalten für q = 1, 2, 3, 4 und für "1. und 3.", "2. und 3." die Werte:

Monatsname	q=1	q=2	q=3	q=4	1.+3.	2.+3.
JANUAR	10	11	25	46	24	15
FEBRUAR	6	11	13	31	8	7
MAERZ	13	14	19	37	18	6
APRIL	1	17	35	44	19	34
MAI	13	14	23	23	22	10
JUNI	10	31	45	54	24	35
JULI	10	31	43	52	22	33
AUGUST	1	22	29	50	8	28
SEPTEMBER	19	24	40	60	35	21
OKTOBER	15	26	46	61	35	31
NOVEMBER	14	29	51	56	36	37
DEZEMBER	4	9	9	14	4	5

Wir verwenden nur die Spalten "q=2", "q=3" und "2.+3.", wählen als p die Zahlen 17 und 22 und erhalten:

Monatsname	q = 2 p=17	q = 3 p=17	2.+3. p=17	q = 2 p=22	q = 3 p=22	2.+3. p=22
JANUAR	11	8	15	11	3	15
FEBRUAR	11	13	7	11	13	7
MAERZ	14	2	6	14	19	6
APRIL	0	1	0	17	13	12
MAI	14	6	10	14	1	10
JUNI	14	11	1	9	1	13
JULI	14	9	16	9	21	11
AUGUST	5	12	11	0	7	6
SEPTEMBER	7	6	4	2	18	21
OKTOBER	9	12	14	4	2	9
NOVEMBER	12	0	3	7	7	15
DEZEMBER	9	9	5	9	9	5

Eine andere Abbildung f erhält man, indem man nicht die ersten q Buchstabenwerte addiert, sondern indem man eine Teilmenge der Indizes {1, 2, ..., r} auswählt und die zugehörigen Buchstabenwerte aufsummiert. In der Tabelle auf den vorherigen Folien sind dies die Teilmengen

{1, 3}, bezeichnet durch 1.+3. sowie
{2, 3}, bezeichnet durch 2.+3.

Die Abbildungen, die in den Spalten angegeben sind, sind untereinander nicht "besser" oder "schlechter", sondern sie sind nur von unterschiedlicher Qualität für unsere spezielle Menge **A** der Monatsnamen. Wir wählen nun irgendeine dieser Funktionen und fügen mit ihr die Monatsnamen in eine Tabelle (= ein array A = Hashtabelle A) mit p Komponenten ein.

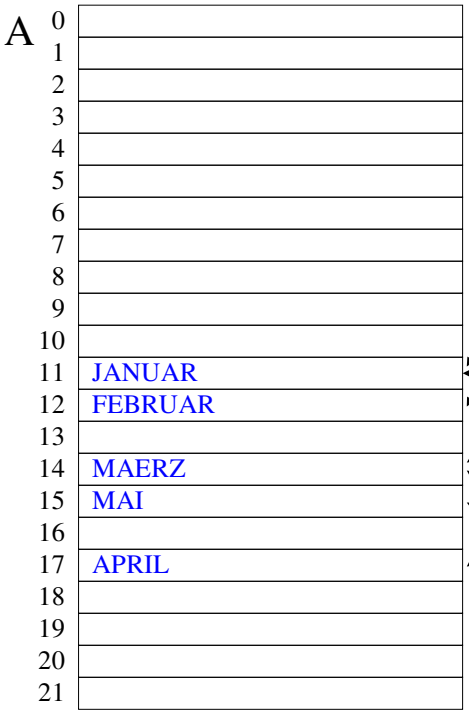
Als Abbildung verwenden wir q=2 und p=22; wir wählen also (willkürlich!) die Hashfunktion

$f(\alpha_{i_1}\alpha_{i_2}...\alpha_{i_r}) = (\varphi(\alpha_{i_1}) + \varphi(\alpha_{i_2}))$ mod 22.

Zum Beispiel ist dann f(JANUAR) = (10 + 1) mod 22 = 11 und f(OKTOBER) = (15 + 11) mod 22 = 4. Alle Werte dieser Abbildung finden Sie in der entsprechenden Spalte für q=2 und p=22 auf der vorletzten Folie.

Die Monatsnamen tragen wir in ihrer jahreszeitlichen Reihenfolge nacheinander in das Feld A ein. Wir nehmen an, dass die Wörter der Menge B höchstens die Länge 20 haben (kürzere Wörter werden durch Zwischenräume, deren φ-Wert 0 sei, aufgefüllt) und deklarieren daher die Hashtabelle:

A: array (0..p-1) of String(20);



Füge das Wort JANUAR mit f(JANUAR) = 11 ein:

Füge das Wort FEBRUAR mit f(FEBRUAR) = 11 ein: **Konflikt!** Verschiebe FEBRUAR um einen Platz nach hinten.

Füge das Wort MAERZ mit f(MAERZ) = 14 ein:

Füge das Wort APRIL mit f(APRIL) = 17 ein:

Füge das Wort MAI mit f(MAI) = 14 ein: **Konflikt!** Verschiebe MAI um einen Platz nach hinten.

Füge nun weiterhin die Wörter JUNI, JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, DEZEMBER ein.

Die zugehörigen f-Werte lauten: 9, 9, 0, 2, 4, 7, 9.

Es entstehen wieder ein **Konflikt** bei JUNI, JULI und DEZEMBER. JULI muss um einen, DEZEMBER um zwei Plätze verschoben werden. Dabei entsteht ein Konflikt mit JANUAR, d.h., man muss DEZEMBER bis Platz 13 verschieben.

A		
	0	AUGUST
	1	
	2	SEPTEMBER
	3	
	4	OKTOBER
	5	
	6	
	7	NOVEMBER
	8	
	9	JUNI
	10	JULI
	11	JANUAR
	12	FEBRUAR
	13	DEZEMBER
	14	MAERZ
	15	MAI
	16	
	17	APRIL
	18	
	19	
	20	
	21	

Dies ist die Hashtabelle nach Einfügen der 12 Namen.

Suchen:
Gesucht wird APRIL. Es ist f(APRIL) = 17. Man prüft, ob A(17) = APRIL ist. Dies trifft zu, also ist APRIL in der Menge.

Gesucht wird JULI. Es ist f(JULI) = 9. Man prüft, ob A(9) = JULI ist. Dies trifft nicht zu, also prüft man, ob A(10) = JULI ist. Dies trifft zu, also ist JULI in der Menge.

A		
	0	AUGUST
	1	
	2	SEPTEMBER
	3	
	4	OKTOBER
	5	
	6	
	7	NOVEMBER
	8	
	9	JUNI
	10	JULI
	11	JANUAR
	12	FEBRUAR
	13	DEZEMBER
	14	MAERZ
	15	MAI
	16	
	17	APRIL
	18	
	19	
	20	
	21	

Gesucht wird DEZEMBER. Es ist f(DEZEMBER) = 9. Man prüft, ob A(9) = DEZEMBER ist, dann für A(10) usw. bis man entweder auf DEZEMBER oder auf einen leeren Eintrag trifft.

Gesucht wird CLAUS. Es ist f(CLAUS) = 15. Man prüft, ob A(15) = CLAUS ist. Dies trifft nicht zu, also geht man zu A(16). Dies ist aber ein leerer Eintrag, also ist CLAUS nicht in der Menge der Monatsnamen.

Wie löscht man? (Später!)

Wie viele Vergleiche braucht man, um einen Namen zu finden, der in der Menge liegt?

JANUAR:	1 Vergleich
FEBRUAR:	2 Vergleiche
MAERZ:	1 Vergleich
APRIL:	1 Vergleich
MAI:	2 Vergleiche
JUNI:	1 Vergleich
JULI:	2 Vergleiche
AUGUST:	1 Vergleich
SEPTEMBER:	1 Vergleich
OKTOBER:	1 Vergleich
NOVEMBER:	1 Vergleich
DEZEMBER:	5 Vergleiche
insgesamt	19 Vergleiche

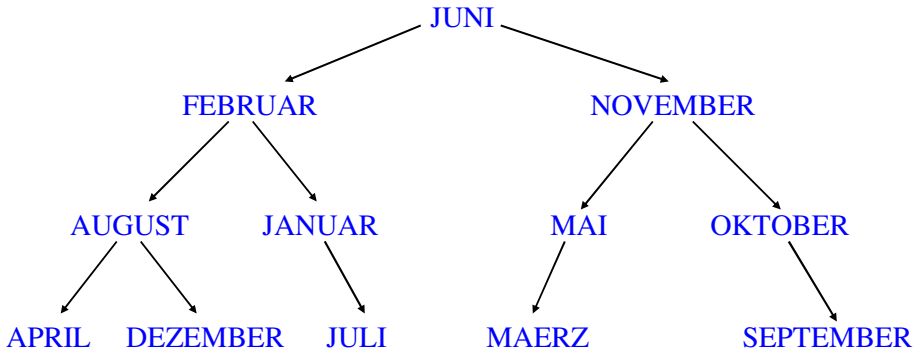
Im Mittel braucht man also 19/12 ≈ **1,6 Vergleiche**, falls der gesuchte Name in der Menge ist.

Wie viele Vergleiche braucht man für einen Namen, der **nicht** in der Menge liegt? Gehe jede Komponente des Feldes hierzu durch (f soll gleichverteilt sein, siehe Anfang von Kapitel 3.3):

0:	2 Vergleiche	11:	6 Vergleiche
1:	1 Vergleich	12:	5 Vergleiche
2:	2 Vergleiche	13:	4 Vergleiche
3:	1 Vergleich	14:	3 Vergleiche
4:	2 Vergleiche	15:	2 Vergleiche
5:	1 Vergleich	16:	1 Vergleich
6:	1 Vergleich	17:	2 Vergleiche
7:	2 Vergleiche	18:	1 Vergleich
8:	1 Vergleich	19:	1 Vergleich
9:	8 Vergleiche	20:	1 Vergleich
10:	7 Vergleiche	21:	1 Vergleich
	Gesamt:		55 Vergleiche

Im Mittel braucht man also 55/22 ≈ **2,5 Vergleiche**, falls der gesuchte Name **nicht** in der Menge ist.

Vergleich mit einem ausgeglichenen Suchbaum:



Mittlere Anzahl der Vergleiche für Elemente, die in der Menge sind: (1+2+2+3+3+3+3+4+4+4+4+4+4) / 12 ≈ **3,1 Vergleiche**.
Falls das Element **nicht** in der Menge ist (13 null-Zeiger):
im Mittel 49 / 13 ≈ **3,8 Vergleiche**.

Wir vernachlässigen hier, dass man an jedem Knoten eigentlich zwei Vergleiche durchführt: auf "Gleichheit" und auf "Größer".

Zeitbedarf: Die Hashtabelle ist deutlich günstiger. Man muss aber die Berechnung der Abbildung f hinzu zählen, die allerdings nur einmal je Wort durchgeführt wird.

Speicherplatz: Wir benötigen 22 statt 12 Bereiche für die Elemente der Menge B. Dafür sparen wir die Zeiger des Suchbaums. Es hängt also vom Platzbedarf ab, den jedes Element aus B braucht, um abschätzen zu können, ob sich diese Tabellendarstellung mit der Abbildung f lohnt.

Sie ahnen es schon: Hashtabellen sind in der Regel deutlich günstiger als Suchbäume. Allerdings darf man die Tabelle nicht zu sehr füllen, da dann die Suchzeiten, insbesondere für Wörter, die *nicht* in der Tabelle sind, stark anwachsen. Erfahrungswert: Mindestens **20%** der Plätze sollten ständig frei bleiben (vgl. Abschnitt 3.3.4).

3.3.2 Hashfunktionen

3.3.2.1 Aufgabe:
Elemente einer Menge B sollen in einem array (0..p-1) of ... gesucht und dort in irgendeiner Reihenfolge eingefügt und gelöscht werden können. Es sei |B| > p (sonst ist die Aufgabe ohne Konflikte durch irgendeine injektive Zuordnung zu lösen).

Benutze hierfür eine Funktion f: B → {0, 1, ..., p-1}, genannt *Hashfunktion,* die surjektiv ist (d.h., jede Zahl von 0 bis p-1 tritt als Bild auf), die die Elemente von B möglichst gleichmäßig über die Zahlen von 0 bis p-1 verteilt und die schnell berechnet werden kann.

In der Praxis verwendet man meist folgende Hashfunktion:

3.3.2.2 Divisionsverfahren (p sollte eine Primzahl sein)

1. Fasse den gegebenen Schlüssel w als Zahl auf (jedes Datum ist binär dargestellt und kann daher als Zahl aufgefasst werden).
2. Bilde den Rest der Division durch die Zahl p
   f(w) = w mod p.

Diese Restbildung hatten wir in 3.3.1 benutzt.

Ein anderes Verfahren verwendet eine "möglichst irrationale" Zahl z zwischen 0 und 1.

3.3.2.3 Multiplikationsverfahren:
(p ist die Größe der Hashtabelle)

1. Fasse wiederum den gegebenen Schlüssel w als Zahl auf.
2. Multipliziere diese Zahl mit z und betrachte nur den Nachkommateil, d.h., die Ziffernfolge nach dem Dezimalpunkt:
   $g(w) = w \cdot z - \lfloor w \cdot z \rfloor$
3. Erweitere dies auf das Intervall [0..p) und bilde den ganzzahligen Anteil:
   $f(w) = \lfloor p \cdot g(w) \rfloor$.

Beispiel: Seien p = 22 und z = 0,624551.

Dann gilt für w = 34: $w \cdot z = 21{,}234734$, $g(w) = 0{,}234734$.

f(w) = ganzzahliger Anteil von $p \cdot g(w) = \lfloor 5{,}164148 \rfloor = 5$.

Hinweise: Die Zahl $|c_2| = 0.6180339887...$ gilt als gut geeignete Zahl z (zu c_2 siehe Fibonaccizahlen nach 3.2.4.6).
Es kann auch $g(w) = \lceil w \cdot z \rceil - w \cdot z$ benutzt werden.

3.3.2.4: Wenn Zeichenfolgen als Schlüsselmenge $B = \Sigma^*$ vorliegen, wählt man gerne ein Teilfolgenverfahren (hier bzgl. der Division vorgestellt; analog: bzgl. der Multiplikation):

1. Codiere die Buchstaben: $\varphi: \Sigma \to \{0, 1, ..., s-1\}$, z.B. ASCII.
2. Wähle fest eine Teilfolge $i_1\ i_2\ ...\ i_q$.
3. Wähle als Hashfunktion $f: \Sigma^* \to \{0, 1, ..., p-1\}$

$$f(\alpha_1\alpha_2 ...\alpha_r) = \left(\sum_{j=1}^{q} \varphi(\alpha_{i_j}) \right) \text{ mod } p$$

oder verwende allgemein eine gewichtete Summe mit irgendwelchen geschickt gewählten Zahlen $a_1, a_2, ..., a_q$:

$$f(\alpha_1\alpha_2 ...\alpha_r) = \left(\sum_{j=1}^{q} a_j \cdot \varphi(\alpha_{i_j}) \right) \text{ mod } p.$$

Definition 3.3.2.5: Eine Hashfunktion heißt perfekt *bzgl. einer Menge* $A \subseteq B$ (mit $|A| \leq p$) von Elementen, wenn f auf der Menge A injektiv ist, wenn also für alle Elemente $a_i \neq a_j$ aus A stets $f(a_i) \neq f(a_j)$ gilt.

Wenn man einen unveränderlichen Datenbestand hat (etwa gewisse Wörter in einem Lexikon oder die reservierten Wörter einer Programmiersprache), so lohnt es sich, eine Hashtabelle mit einer perfekten Hashfunktion einzusetzen, da dann die Entscheidung, ob ein Element b in der Tabelle vorkommt, durch eine Berechnung f(b) und einen weiteren Vergleich getroffen werden kann.

Durch Ausprobieren lassen sich oft solche perfekten Funktionen finden. Suchen Sie z.B. eine für **A** = {JANUAR, ... , DEZEMBER} und p=15.

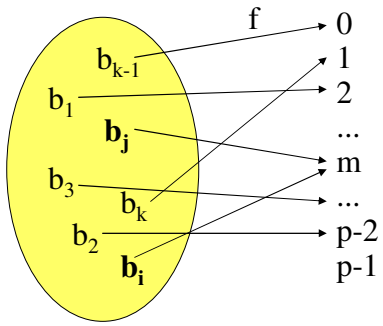
Eine Lösung lautet für **A** = {JANUAR, ... , DEZEMBER} und p=15:

$f(\alpha_1\alpha_2 \ldots\alpha_r) = \left(7\,\varphi(\alpha_1) + 5\,\varphi(\alpha_2) + 2\,\varphi(\alpha_3)\right) \underline{\text{mod}}\ 15.$

Dieses f ist tatsächlich injektiv:

JANUAR	13
FEBRUAR	11
MAERZ	1
APRIL	3
MAI	9
JUNI	8
JULI	4
AUGUST	6
SEPTEMBER	10
OKTOBER	5
NOVEMBER	7
DEZEMBER	0

3.3.2.6: Wahrscheinlichkeit für einen Konflikt.
In eine Tabelle von p Plätzen sollen nun k Elemente aus B mit Hilfe einer Hashfunktion $f: B \rightarrow \{0, 1, ..., p-1\}$ eingetragen werden. Eine Hashfunktion f soll die Elemente aus B möglichst gleichmäßig auf die p Zahlen abbilden. Wie groß ist die Wahrscheinlichkeit, dass unter k verschiedenen Elementen mindestens zwei Elemente b_i und b_j sind mit $f(b_i) = f(b_j)$?



Berechne die Wahrscheinlichkeit, dass unter k verschiedenen Elementen mindestens zwei Elemente b_i und b_j sind mit $f(b_i)=f(b_j)$. Dies ist 1 minus der Wahrscheinlichkeit, dass alle k Elemente auf verschiedene Werte abgebildet werden:

$$1 - (1 - \frac{1}{p}) \cdot (1 - \frac{2}{p}) \cdot \ldots \cdot (1 - \frac{k-1}{p})$$

$$= 1 - \prod_{i=1}^{k-1} e^{-\frac{i}{p}} \approx 1 - e^{-\frac{k(k-1)}{2p}}$$

Beachte hierbei: $(1-i/p) \approx e^{-\frac{i}{p}}$

Wann beträgt die Wahrscheinlichkeit 50%, dass mindestens zwei Schlüssel auf den gleichen Wert abgebildet werden?

$1 - e^{-\frac{k(k-1)}{2p}} = ½$ liegt vor bei

$\ln(½) = -\frac{k(k-1)}{2p}$, d.h., es gilt ungefähr

$k \approx \sqrt{p \cdot 2\ln(2)}$ mit $2\ln(2) \approx 1{,}386$ und $\sqrt{2\ln(2)} \approx 1{,}1777$.

Satz 3.3.2.7

Trägt man gleichverteilte Schlüssel nacheinander in eine Hashtabelle der Größe p ein, so muss man nach $1{,}1777 \cdot \sqrt{p}$ Schritten damit rechnen, dass "Kollisionen" eintreten, dass also zwei verschiedene Schlüssel auf den gleichen Platz eingetragen werden wollen.

3.3.2.8: Wie viele verschiedene Plätze der Hashtabelle werden im Mittel durch f(b) angesprochen, wenn man k verschiedene Schlüssel b nacheinander betrachtet?
Hierzu lösen wir zunächst die Frage, wie groß die Wahrscheinlichkeit ist, dass ein Platz hierbei nicht besucht wird. Diese Wahrscheinlichkeit ist:

$$(1-1/p)^k \approx e^{-\frac{k}{p}} = e^{-\lambda}$$

mit $\lambda = k/p$ "Auslastungsgrad" der Tabelle.

Für $\lambda = 1$ wird jeder Platz also mit der Wahrscheinlichkeit $(1-e^{-1}) \approx 0{,}63212...$ besucht. Das heißt, wenn man p Elemente nacheinander in eine Hashtabelle der Größe p einfügt, so werden hierbei rund 63,2% verschiedene Tabellen-Indizes beim Ausrechnen der Hashfunktion berechnet. Es treten also viele Kollisionen auf.

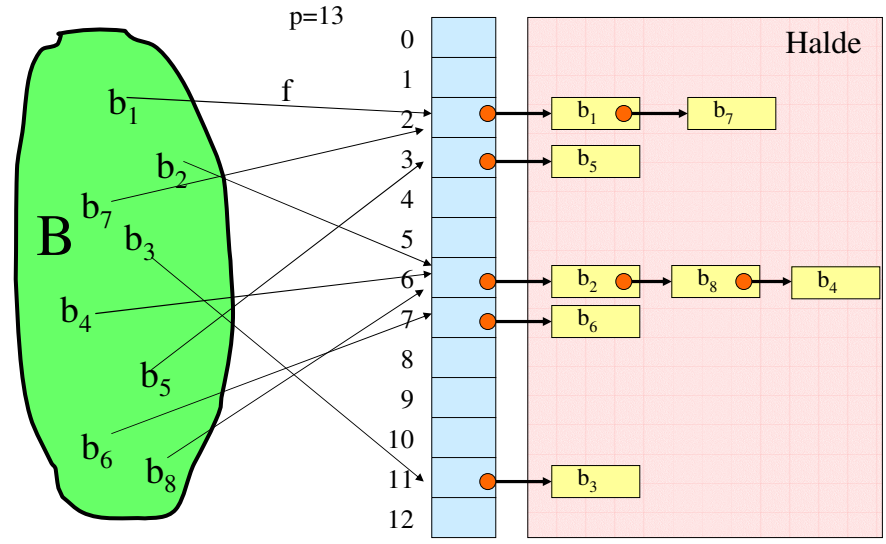
3.3.2.9 Aufbau einer Hashtabelle (mit externer Kollision)

Alle Schlüssel, die den gleichen f-Wert haben, werden in einer linearen Liste gespeichert. In der Hashtabelle A steht an der Stelle A(i) der Zeiger auf die Liste der Schlüssel b mit f(b) = i.

Auf diese Weise entstehen keine Kollisionen in der Hashtabelle, sondern dieses Problem wird in die Haldenverwaltung verlagert, die die bis zu p Listen zu organisieren hat.

Dieses Vorgehen erfordert einen Zugriff auf das Feld A(f(b)) und anschließend muss eine lineare Liste durchsucht werden. Der Vorteil ist, dass keine feste obere Grenze für die Menge der Schlüssel **A** vorgegeben werden muss, da in der Halde meist viel Platz ist. Der Nachteil besteht in der Abhängigkeit von der Haldenverwaltung (Zugriffszeiten, Garbage Collection). Daher verwendet man in der Praxis meist andere Verfahren, vor allem das "offene Hashing", siehe 3.3.3.

Skizze: "Externe" Hashtabelle



Überlaufprobleme müssen mit der Haldenverwaltung gelöst werden!

3.3.3 Offenes Hashing

Die Schlüssel werden in der Hashtabelle selbst gespeichert. Wenn eine Kollision auftritt, wird ein neuer Platz gesucht. Es erfolgt nun eine Kollisionsstrategie. Wird dabei der zweite Schlüssel auf dem ersten freien ("offenen") Platz, den man nach dieser Kollisionsstrategie erreicht, abgelegt, so spricht man von "offenem Hashing".

Das Suchen geht in der Regel schnell, sofern mindestens 20% der Plätze des array frei gehalten werden.

Das Einfügen ist nicht schwierig.

Problem: Löschen.

3.3.3.1 Datentypen festlegen (die Booleschen Werte brauchen wir erst später):

```
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;
```

FIND: Der Suchalgorithmus lautet dann: ...

INSERT: Der Einfügealgorithmus lautet dann: ...

3.3.3.2 Einfüge-Algorithmus

```
A: hashtabelle; i, j: integer;              --  p sei global bekannt
k: integer :=0;       -- k gibt die Anzahl der Schlüssel in A an
for i in 0..p-1 loop  A(i).besetzt:=false; A(i).kollision:=false;
      A(i).geloescht:=false; A(i).behandelt:=false; end loop;
while "es gibt noch einen einzutragenden Schlüssel b" loop
if k < p then
      k := k+1; j := f(b);          -- j ist die Adresse in A für b
      if not A(j).besetzt or A(j).geloescht then  A(j).besetzt := true;
            A(j).Schluessel := b; A(j).inhalt := ...;
      else A(j).kollision := true; "Starte eine Kollisionsstrategie";
      end if;
else "Tabelle A ist voll, starte eine Erweiterungsstrategie für A";
end if;
end loop;
```

3.3.3.3 Das Suchen erfolgt ähnlich:

Um einen Eintrag mit dem Schlüssel b zu finden, berechne f(b) und prüfe, ob in A(f(b)) ein Eintrag mit dem Schlüssel b steht. Falls ja, ist die Suche erfolgreich beendet, falls nein, prüfe A(f(b)).kollision. Ist dieser Wert false, dann ist die Suche erfolglos beendet, anderenfalls gehe mit der verwendeten Kollisionsstrategie (s.u.) zu einem anderen Platz A(j) und prüfe erneut, ob der Schlüssel b dort steht, falls nein, welchen Wert A(j).kollision hat usw.

Wir wenden uns nun den Kollisionen und ihrer Behandlung zu.

Definition 3.3.3.4: Sei f: B → {0, 1, ..., p-1} eine Hashfunktion.

Gilt f(b) = f(b') für zwei einzufügende Schlüssel b und b', so spricht man von einer Primärkollision. In diesem Fall muss der zweite Schlüssel b' an einer Stelle A(i) gespeichert werden, für die i ≠ f(b') gilt.

Ist f(b') = i und befindet sich auf dem Platz A(i) ein Schlüssel s mit f(s) ≠ i, so spricht man von einer Sekundärkollision, d.h., die erste Kollision beim Eintragen von b' wird durch einen Schlüssel s verursacht, der selbst durch eine Kollision an diese Position f(b') gelangt ist.

Wenn man eine Strategie zur Behandlung von Kollisionen festlegt, so kann man sich gegen die Primärkollisionen kaum wehren, aber man kann versuchen, die Sekundärkollisionen klein zu halten.

Beispiel: Sei
A = {JANUAR, FEBRUAR, MAERZ, APRIL, MAI, JUNI, JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, DEZEMBER} mit $f(\alpha_1\alpha_2 ...\alpha_r) = (\varphi(\alpha_1) + \varphi(\alpha_2))$ mod 14. Drei Schlüssel werden in der Reihenfolge JANUAR, FEBRUAR, OKTOBER eingegeben. Es gilt:
f(JANUAR) = 11, f(FEBRUAR) = 11, f(OKTOBER) = 12.

Wir tragen JANUAR im Platz A(11) ein.
Der Schlüssel FEBRUAR führt zu einer Primärkollision. Die Strategie möge lauten: *Gehe von Platz j zum nächsten Platz j+1*. Dann wird FEBRUAR in dem Platz A(12) gespeichert.
Der Schlüssel OKTOBER gehört in den Platz A(12), doch hier steht ein Schlüssel, der dort durch eine Kollision hinverschoben wurde. Folglich führt OKTOBER zu einer Sekundärkollision. OKTOBER wird dann auf Platz A(13) eingetragen.

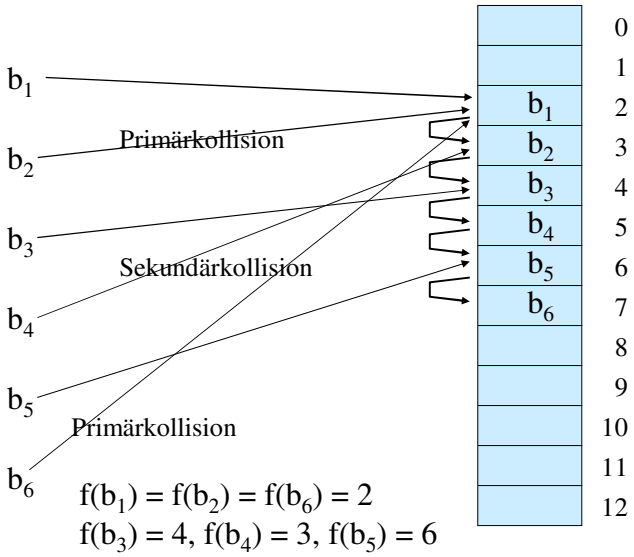
Definition 3.3.3.3: Kollisionsstrategien

Wenn auf Platz j eine Kollision stattfindet, so versuche man, den Schlüssel b auf dem Platz G(...) einzufügen. Es sei i die Zahl der versuchten Zugriffe. c ist eine fest gewählte Konstante; man kann hier stets c=1 wählen; wichtig ist ggT(c,p)=1.

G(j) = (j+c) mod p heißt "lineare Fortschaltung" oder "lineares Sondieren" oder "Lineares Hashing".

$G(b,i) = (f(b)+i^2)$ mod p heißt "quadratische Fortschaltung" oder "quadratisches Sondieren".

Beispielskizze: Lineares Sondieren mit c=1 p=13



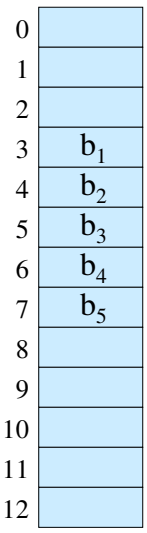
Das lineare Sondieren ist ein leicht zu implementierendes Verfahren, das sich in der Praxis bewährt hat. *Nachteil* ist die sog. "Clusterbildung", die durch Sekundärkollissionen hervorgerufen wird (siehe auch das vorige Beispiel):
Die Wahrscheinlichkeit, auf eine bereits durchlaufene Kollisionkette zu stoßen, wird im Laufe der Zeit größer als die Wahrscheinlichkeit, auf Anhieb einen freien Platz zu finden. Dadurch werden die Kollisionketten immer nachvollzogen: Es bilden sich sog. Cluster, siehe Erläuterung unten.
Der *Vorteil* des linearen Sondierens liegt darin, dass man noch brauchbare Verfahren hat, um Werte aus der Hash-tabelle wieder zu löschen. (Selbst überlegen; nicht schwer. Das prinzipielle Vorgehen wird unten erläutert.)

Das quadratische Sondieren ist ebenfalls leicht zu implementieren. Sein *Vorteil* ist, dass die Clusterbildung durch Sekundärkollisionen vermieden werden. (Bei Primärkollisionen müssen natürlich alle Versuche erneut nachvollzogen werden.)

Der *Nachteil* des quadratischen Sondierens besteht darin, dass der Aufwand, um Werte aus der Hashtabelle wieder zu löschen, zu groß ist. Man markiert daher die gelöschten Elemente als "gelöscht", belässt sie aber weiter in der Tabelle und entfernt alle gelöschten Elemente erst nach einiger Zeit gemeinsam (siehe unten).

Um nicht in Zyklen bei der Kollisionsstragegie zu gelangen, soll man beim quadratischen Sondieren unbedingt verlangen, dass die Größe der Hashtabelle p eine Primzahl ist. Dies wird im Folgenden begründet.

3.3.3.4 Clusterbildung bei linearem Sondieren



Es möge die nebenstehende Situation mit dem Cluster A(3) bis A(7) entstanden sein.

Es soll nun ein weiterer Schlüssel b eingefügt werden. Ist f(b) einer der Werte 3, 4, 5, 6 oder 7, so wird b bei linearem Sondieren mit c=1 in A(8) gespeichert.

Die Wahrscheinlichkeit, dass im nächsten Schritt A(8) belegt wird, ist daher 6/13, während für jeden anderen Platz nur die Wahrscheinlichkeit 1/13 gilt.

Cluster haben also eine hohe Wahrscheinlichkeit, sich zu vergrößern. Genau dieser Effekt wird in der Praxis beobachtet.

Die Clusterbildungen beruhen auf den Sekundärkollisionen. Diese werden beim quadratischen Sondieren vermieden.

Als Beispiel betrachten wir erneut **A** = {JANUAR, FEBRUAR, MAERZ, APRIL, MAI, JUNI, JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, DEZEMBER} mit p=22.

Als Abbildung verwenden wir dieses Mal die Hashfunktion

$f(\alpha_1\alpha_2 \ldots\alpha_r) = (2\varphi(\alpha_1) + \varphi(\alpha_2))$ mod 17.

A

0	FEBRUAR
1	APRIL
2	
3	JANUAR
4	
5	
6	AUGUST
7	JUNI
8	JULI
9	SEPTEMBER
10	MAERZ
11	MAI
12	
13	NOVEMBER
14	DEZEMBER
15	
16	OKTOBER

Quadratisches Sondieren: Wir fügen die Wörter ein JANUAR, FEBRUAR, MAERZ, APRIL, MAI, JUNI, JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, DEZEMBER .

Die zugehörigen f-Werte lauten: 4, 0, 10, 1, 10, 7, 7, 6, 9, 7, 9, 13.

Tragen Sie die Wörter ein. Es ergibt sich die nebenstehende Tabelle.

3.3.3.5: Länge von Zyklen bei Kollisionsstrategien

Wir müssen uns nun überzeugen, dass bei den Kollisionsstrategien keine zu kleinen Zyklen durchlaufen werden. Beim linearen Sondieren ist dies gewährleistet: Wenn c und p teilerfremd sind (ggT(c,p)=1), dann durchläuft die Folge der Zahlen $(j+c) \bmod p$ (für j = 0, 1, 2, ...) alle Zahlen von 0 bis p-1, bevor eine Zahl erneut auftritt.

Wir wollen nun zeigen, dass beim quadratischen Sondieren keine "kurzen" Zyklen auftreten, sofern p eine Primzahl ist.

Wir fragen daher: Wann tritt in der Folge der Zahlen

$$(f(b)+i^2) \bmod p \quad \text{für } i = 0, 1, 2, 3, ...$$

erstmals eine Zahl wieder auf?

Wenn eine Zahl erneut auftritt, so muss es zwei Zahlen i und j geben mit $i \neq j$, $i \geq 0$, $j \geq 0$ und

$(f(b)+i^2) \bmod p = (f(b)+j^2) \bmod p$,

d.h. $(i^2-j^2) \bmod p = (i+j)\cdot(i-j) \bmod p = 0$.

Wenn p eine Primzahl ist, dann muss (i-j) oder (i+j) durch p teilbar sein. Wir nehmen an, dass wir höchstens p mal das quadratische Sondieren durchführen, d.h., dass $0 \leq i \leq p-1$ und $0 \leq j \leq p-1$ gelten. Dann ist $-p < (i-j) < p$ und wegen $i \neq j$ kann daher p nicht (i-j) teilen. Also muss p die Zahl (i+j) teilen. Das geht aber nur, wenn mindestens eine der beiden Zahlen größer als die Hälfte von p+1 ist. Also gilt:

Satz 3.3.3.6:
Beim quadratischen Sondieren kann frühestens nach (p+1)/2 Schritten eine Zahl erneut auftreten, sofern p eine Primzahl ist.

Für i = (p+1)/2 und j = (p-1)/2 gilt: (i+j)·(i-j) mod p = 0. Die in Satz 3.3.3.6 genannte kürzeste Länge eines Zyklus von (p+1)/2 Schritten tritt in der Tat für *alle* Primzahlen auf, wie man sich sehr leicht klar macht. Die jeweils "komplementären Zahlen" $(-i^2)$ mod p kommen dagegen in diesem Zyklus nicht vor. Modifiziert man daher das quadratische Sondieren so, dass zwischen i^2 mod p und $(i+1)^2$ mod p immer $(-i^2)$ mod p eingeschoben wird, so erreicht man die volle Zykluslänge p.

Aufgabe:
Beweisen Sie diese Aussage und schreiben Sie eine Prozedur für diese Vorgehensweise.

Tritt beim linearen oder beim quadratischen Sondieren eine Primärkollision (= zwei verschiedene Schlüssel haben den gleichen Hashwert) auf, so wird für das Einfügen des jeweils letzten Schlüssels die gesamte Kette der Kollisionen, die die früheren Schlüssel mit gleichem Hashwert durchlaufen haben, ebenfalls durchlaufen.

Will man diesen Effekt vermeiden, so muss man eine zweite Hashfunktion g hinzunehmen, die möglichst unabhängig von f ist, d.h., für f und g sollte auf jeden Fall gelten:

$B_{m,n} = \{b \in B \mid f(b) = m \text{ und } g(b) = n\}$ enthält für alle m und n ungefähr $|B|/p^2$ Elemente.

Dies führt zu "Doppel-Hash"-Kollisionsverfahren:

Definition 3.3.3.7: Kollisionsstrategien (Fortsetzung)

Es seien f und g zwei unterschiedliche Hashfunktionen. Sei i die Zahl der Zugriffe. Die Kollisionsstrategie

$G(i,b) = (f(b) + i \cdot g(b)) \bmod p$ heißt "Doppel-Hash-Verfahren".

Es seien $f_1, f_2, f_3, f_4, \ldots$ eine Folge von möglichst unterschiedlichen Hashfunktionen. Die Kollisionsstrategie

$G(i,b) = f_i(b)$ heißt "Multi-Hash-Verfahren".

Hinweis: In der Praxis hat man mit Doppel-Hash-Strategien gute Erfahrungen gemacht.

Löschen in Hashtabellen (DELETE)
Dieses bildet das Hauptproblem in Hashtabellen.

Der einfachste Weg ist es, das Löschen durch Setzen eines Booleschen Wertes zu realisieren: Wenn der Eintrag mit dem Schlüssel b gelöscht werden soll, so suche man seine Position A(j) auf und setze A(j).geloescht := true.
Beim Einfügen behandelt man dieses Feld A(j) dann wie einen freien Platz.

Nachteil: Wenn oft gelöscht wird, dann ist die Tabelle schnell voll und muss mit gewissem Aufwand reorganisiert werden, vgl. Abschnitt 3.3.5. Dennoch ist dieses Vorgehen in der Praxis gut einsetzbar.

Hat man sich jedoch für das lineare Sondieren entschieden, dann kann man das Löschen korrekt durchführen: Man sucht den Eintrag A(j) mit dem zu löschenden Element auf und geht dann die Einträge A(j+c), A(j+2c), A(j+3c) solange durch, bis man auf einen freien Platz stößt. In dieser Kette kopiert man alle Einträge um c, 2c, 3c usw. Plätze zurück, aber niemals über den Platz k hinaus mit f(b)=k.

Details: selbst überlegen! Siehe auch Übungen.

3.3.4 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.

3.3.4.1 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 ($1 \leq i \leq k+1$).

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 lässt sich nicht allzu schwer herleiten. Versuchen Sie es selbst einmal.)

Satz 3.3.4.2: (Beachte die Annahmen 3.3.4.1.)

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 $\mathbf{E_{k+1}} = \frac{p+1}{p+1-k}$

λ	$\mathbf{E_{k+1}}$	λ	$\mathbf{E_{k+1}}$	λ	$\mathbf{E_{k+1}}$
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 soll evtl. in den Übungen ausgerechnet werden.)

Satz 3.3.4.3: (Beachte die Annahmen 3.3.4.1.)

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

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

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

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 $\lambda = k/(p+1)$:

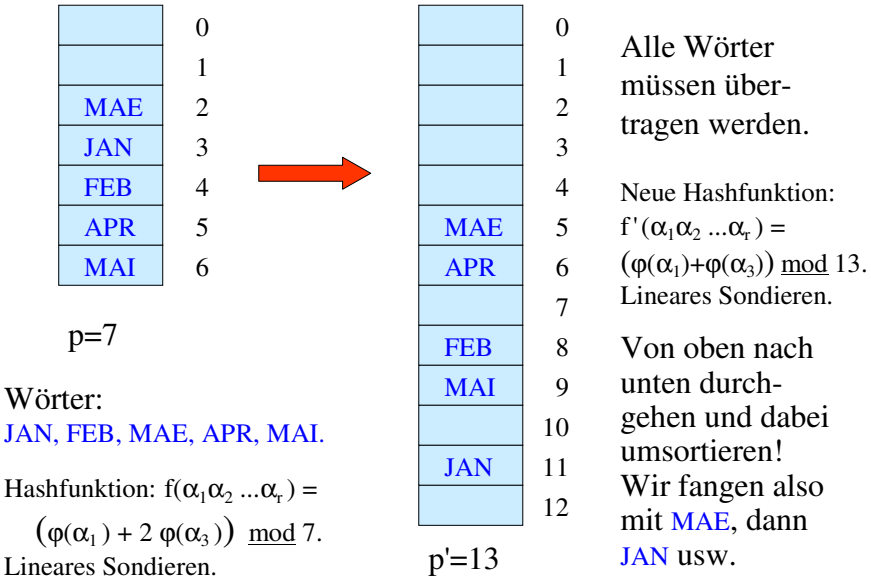
λ	S_k	$Slin_k$
0,50	1,39	1,50
0,75	1,85	2,50
0,80	2,01	3,00
0,90	2,56	5,50
0,95	3,15	10,50
0,99	4,65	50,50

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

3.3.5 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.

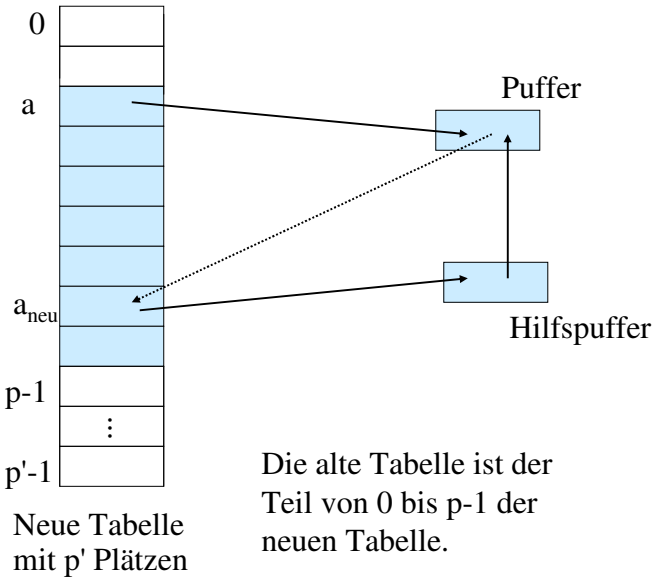
Diesen Vorgang der Umorganisation innerhalb der bestehenden Hashtabelle bezeichnen wir als "Rehashing". Dieses Verfahren wird auch verwendet, wenn man Schlüssel, statt sie zu löschen, nur als "gelöscht" markiert, wodurch im Laufe der Zeit der Auslastungsgrad zu groß wird und eine Umorganisation mit dem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



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:



Man kommt also mit zwei Zusatzvariablen "Puffer" und "Hilfspuffer" aus, in denen die gerade betrachteten oder die weiter zu verschiebenden Elemente zwischengespeichert werden. Nun zur Programmierung:

Erinnerung: (siehe 3.3.3.1)

```
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;
```


Programm für das Rehashing

Bevor die Hashtabelle erstmals benutzt wird, 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 der Verwendung der Tabelle A wurden A(j).besetzt, A(j).kollision und A(j).geloescht eventuell verändert.

Erforderliche Variablen:
Puffer, Hilfspuffer: Eintragtyp;

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

Die verwendete Hashfunktion f' möge zwei Parameter haben: den Schlüssel und die Anzahl i der Zugriffe (= die Anzahl der bisherigen Kollisionen).

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 in diesem Fall mit der neuen Hashfunktion f' den neuen Index a_{neu}, wohin das Element des Puffers hingehört.

(3) Unterscheide hierzu 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 in Ada 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.
```

Programmstück in Ada zum Rehashing

```
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;
```

Zeitaufwand hierfür? (Selbst durchdenken.)