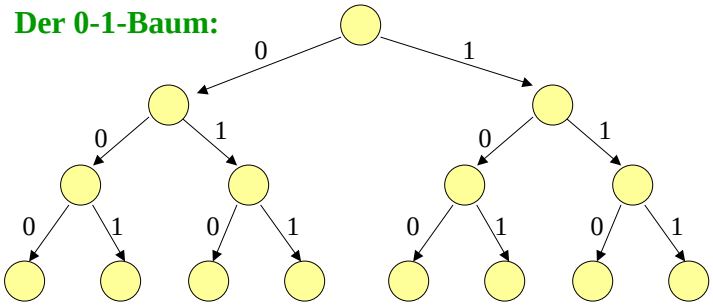
4.4 Digitale Suchbäume

Der 0-1-Baum:



In einem unendlichen Baum, dessen Kanten mit 0 oder 1 markiert sind, hat jede 0-1-Folge seinen eindeutigen Platz.

Digitaler Suchbaum:

Gegeben sind Schlüssel, die 0-1-Folgen sind. Für jeden solchen Schlüssel durchlaufe man den 0-1-Baum und platziere den Schlüssel auf den ersten freien Knoten.

Bedingung: Die Länge aufeinander folgender 0-1-Schlüssel darf nicht kürzer werden!

FIND, INSERT und DELETE verhalten sich dann wie bei binären Suchbäumen.

Vorteile vor allem, wenn man auf Maschinenebene programmiert oder die Binärdarstellung von Schlüsseln vorliegt.

Suche nach dem Knoten mit Schlüssel v

```
hilf1: NodeRef;  b: integer := maxb;
begin  hilf1 := „Verweis auf den digitalen Suchbaum“ ;
      while hilf1 /= null and the hilf1.key /= v loop
            if bits(v,b,1) = 0 then hilf1:=hilf1.l;
                              else hilf1:=hilf1.r; end if;
            b:= b-1;
      end loop;
end;
```

Der Datentyp Node hat Komponenten
key und die Nachfolgerzeiger l und r.


```
function digitalinsert (v: integer; x: NodeRef)
            return Noderef is
hilf1, hilf2: NodeRef;  b: integer := maxb;
begin  hilf1 := x;  hilf2 := null;
      while hilf1 /= null and then hilf1.key /= v loop
            hilf2 := hilf1;
            if bits(v,b,1) = 0 then hilf1:=hilf1.l;
                              else hilf1:=hilf1.r; end if;
            b:= b-1;
      end loop;
      if hilf1 = null then        -- neuen Knoten einfügen
            hilf1 := new Node'(key => v, l => null, r => null);
            if bits(v,b+1,1) = 0 then hilf2.l := hilf1;
                              else hilf2.r:=hilf1; end if;
      end if;
      return hilf1;
end;
```

< Ab Knoten x nach Schlüssel v suchen und evtl. einfügen. Der Verweis auf den Knoten mit dem Schlüssel v wird zurückgegeben>

5. Hashing (gestreute Speicherung)

Vgl. auch Skript Plödereder, SS 01, Folien 178 bis 202

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 (vgl. Anfang zu Kapitel 4): 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**.

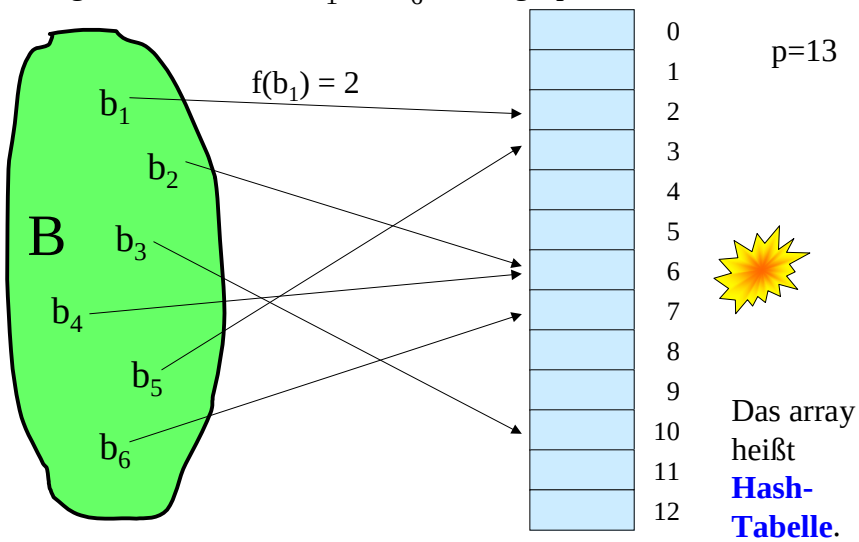
Nehmen wir an, wir hätten eine solche Abbildung
f: B $\rightarrow$ {0, 1, ..., p-1}, dann würden wir zur Speicherung von Teilmengen von B ein Feld deklarieren:
A: array (0..p-1) of <Datentyp für die Menge B>

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?

5.1 Beispiel "modulo p"

Das Problem hängt wesentlich von der Menge B ab.

In der Praxis ist B oft ein freies Monoid, d.h., es sei $B = \Sigma^*$ = die Menge aller Folgen über einem s-elementigen Alphabet $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_s\}$.

Wir legen diese Menge Σ^* (oder eine geeignete Teilmenge) für das Folgende zugrunde.

Weiterhin sei p eine natürliche Zahl, p>1.

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

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

Wir probieren dies am lateinischen Alphabet aus, wobei wir nur die großen Buchstaben A, B, C, ... verwenden. Als Codierung φ wählen wir die Stelle des Buchstabens im Alphabet:

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	26

Als abzubildende Menge **A** wählen wir die Monatsnamen:

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	35	40	30	31

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	1	14	9	13	9

Eine andere Abbildung f erhält man, indem man nicht die ersten q Buchstabenwerte addiert, sondern indem man eine 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

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

Die Werte dieser Abbildung finden Sie in der entsprechenden Spalte für q=2 und p=22 in Folie 12.

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

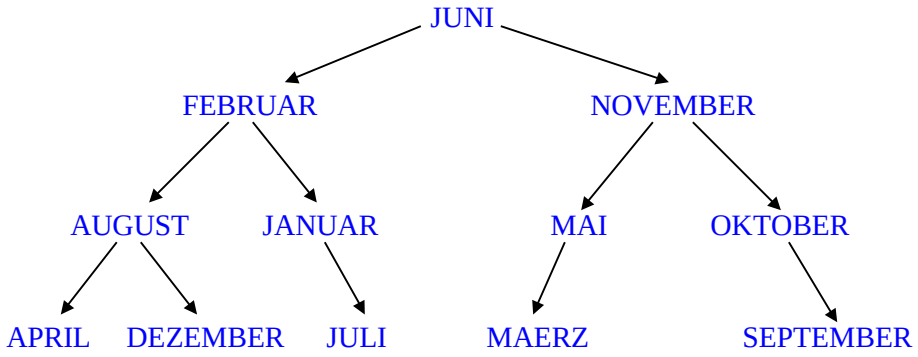
Im Mittel braucht man also
19/12 ≈ **1,6 Vergleiche**, falls der gesuchte Name in der Menge ist.

Wie viele Vergleiche braucht man, um für einen Namen, der **nicht** in der Menge liegt, dies festzustellen? Gehe jede Komponente des Feldes hierzu durch (f soll gleichverteilt sein, Folie 2):

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 55/21 ≈ **2,6 Vergleiche**, falls der gesuchte Name **nicht** in der Menge ist.

Vergleich mit einem möglichst gleichverzweigten Suchbaum:



Mittlere Anzahl der Vergleiche für Elemente, die in der Menge sind: (1+2+2+3+3+3+3+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 5.4).

5.2 Hashfunktionen

Aufgabe: Elemente einer Menge B sollen in ein
array (0..p-1) of ... in irgendeiner Reihenfolge eingefügt, gesucht und dort wieder gelöscht werden. Benutze hierfür eine Hashtabelle mit einer Hashfunktion.

In der Praxis verwendet man oft folgende Hashfunktion:

Divisionsverfahren (p sollte hierbei eine Primzahl sein)

1. Fasse den bisherigen Schlüssel 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
   $f(w) = w \bmod p$.

Umgekehrt kann man p auch mit einer Zahl zwischen 0 und 1 multiplizieren. Dies führt zum Multiplikationsverfahren, wobei man eine Zahl β mit $0 \le \beta < 1$ fest wählt (hierfür eignet sich z.B. die Zahl $(\sqrt{5}-1)/2 \approx 0{,}618033...$, vgl. Fibonaccizahlen):

1. Fasse den bisherigen Schlüssel als Zahl s auf.
2. Berechne $\beta \cdot s$ und nimm den Nachkommateil ν_s dieser Zahl.
3. Setze f(s) = ganzzahliger Anteil von $p \cdot \nu_s$.

Beispiel: Seien p = 22 und $\beta = 0{,}624551$. Dann gilt für s = 34:
$\beta \cdot s = 21{,}234734$, $\nu_s = 0{,}234734$,
f(s) = ganzzahliger Anteil von $p \cdot \nu_s = \lfloor 5{,}164148 \rfloor = 5$.

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

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

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

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

Eine Hashfunktion heißt perfekt bzgl. einer Menge $A \subset B$ 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.

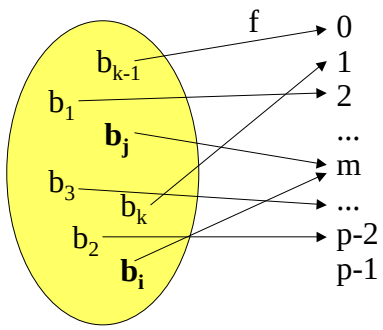
Eine Lösung lautet für **A** = {JANUAR, ... , DEZEMBER} und p=15:

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

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

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 $\sqrt{2 \ln(2)} \approx 1{,}1777$.

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.

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 nur 63,212% verschiedene Tabellen-Indizes beim Ausrechnen der Hashfunktion berechnet. Es treten also viele Kollisionen auf.

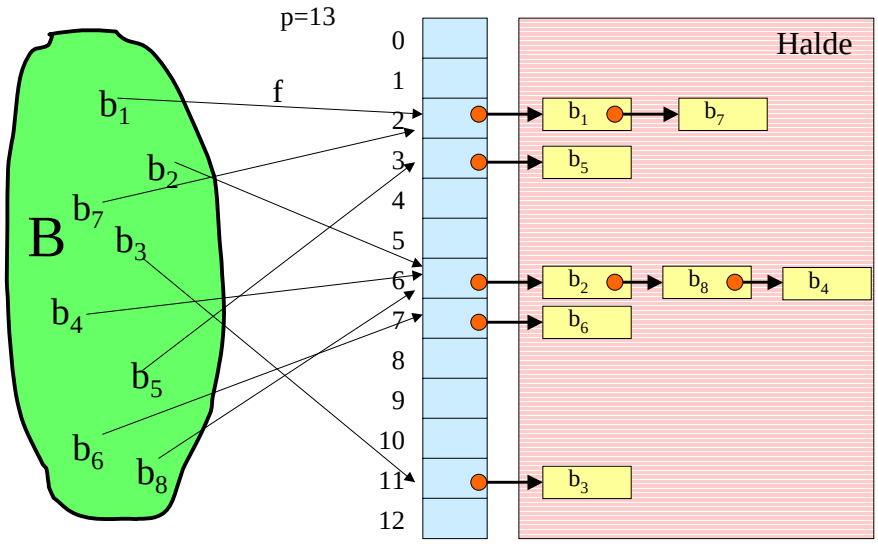
Aufbau einer Hashtabelle (mit externer Kollision)

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.

Erfahrung: Nach einiger Zeit wird der Umorganisationsaufwand in der Halde relativ groß, es sei denn, man spendiert viel freien Platz in der Halde. Daher verwendet man in der Praxis meist andere Verfahren, vor allem das "offene Hashing".

Skizze: Wir greifen ein früheres Bild einer Hashtabelle auf:



Überlaufprobleme müssen mit der Haldenverwaltung gelöst werden!

5.3 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 macht meist keine Probleme. Jedoch lasssen sich Schlüssel nur im Falle des Linearen Sondierens (s.u.) einigermaßen leicht löschen.

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;
```

INSERT: Der Einfügealgorithmus lautet dann:

INSERT

```
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;
```

FIND: 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: Sei f: B $\rightarrow$ {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 anderen Stelle A(i) gespeichert werden, für die i $\neq$ f(b') gilt.

Ist f(b') = i und befindet sich auf dem Platz A(i) ein Schlüssel b mit f(b) $\neq$ i, so spricht man von einer Sekundärkollision.

Wenn man eine Strategie zur Behandlung von Kollisionen festlegt, so kann man sich gegen die Primärkollisionen nicht 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.

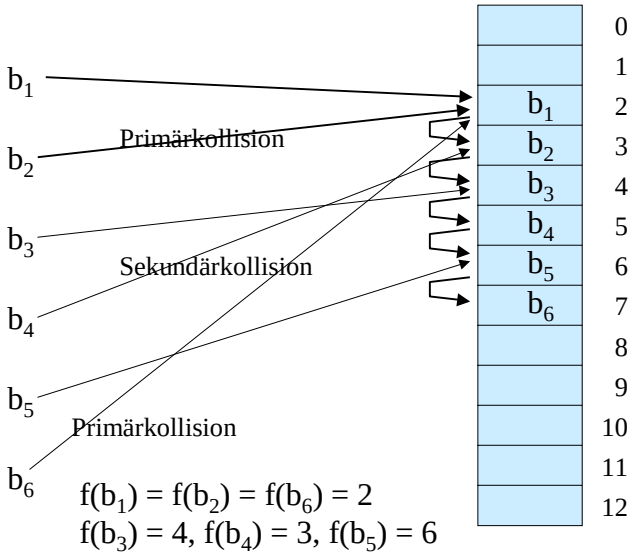
Definition: Kollisionsstrategien

Wenn auf Platz j eine Kollision statt findet, 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.

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

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

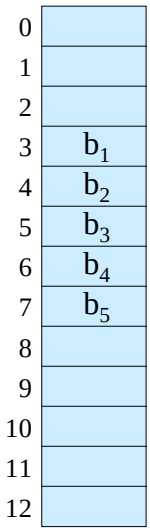
Beispielskizze: Lineares Sondieren mit c=1 p=13



Beim quadratischen Sondieren wird zunächst versucht, den Schlüssel b auf Platz f(b) abzulegen. Ist dieser Platz besetzt, so probiert man den nächsten Platz (f(b) + 1); ist auch dieser Platz besetzt, so geht man drei Plätze weiter zum Platz (f(b)+4), danach fünf Plätze weiter zum Platz (f(b) + 9) usw.

Hierdurch vermeidet man die sog. "Clusterbildung", die typisch für das Lineare Sondieren ist. Ein "Cluster" ist hierbei eine Folge von besetzten Plätzen, von der eine Teilfolge bis zum letzten Platz durchlaufen werden muss, wenn man mittels f(b) auf irgendeinen der Plätze des Clusters am Anfang zugreift. Die Clusterbildung entsteht durch das lineare Fortschalten.

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 $f(\alpha_1\alpha_2 ...\alpha_r) = \left(2\varphi(\alpha_1) + \varphi(\alpha_2)\right) \bmod 17$.

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.

Wir müssen uns nun überzeugen, dass bei den Kollisionsstrategien keine zu kleinen Zyklen durchlaufen werden. 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 man beim quadratischen Sondieren nicht in "kurze" Zyklen geraten kann, sofern p eine Primzahl ist. Wir fragen daher: Wann tritt in der Folge der Zahlen

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

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:

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

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 Kollision, 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: 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.

DELETE : Zum Löschen 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 diesen Eintrag A(j) auf und setze A(j).geloescht := true.
Beim Eintragen behandele man diesen Eintrag A(j) dann wie einen freien Platz, siehe Programm auf Folie 36.

Nachteil: Wenn oft gelöscht wird, dann ist die Tabelle schnell voll und muss mit größerem Aufwand reorganisiert werden, vgl. Abschnitt 5.5. Dennoch ist dieses Vorgehen in der Praxis gut einsetzbar.

Hat man sich jedoch für das lineare Sondieren entschieden, dann kann man das Löschen korekt 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!