H. Hashing

H.1 Einführung (am Beispiel)
H.2 Hashfunktionen
H.3 Techniken beim Hashing
H.4 Rehashing
H.5. Beispiel

Ziel dieses H. Kapitels:

Beim Hashing werden die Elemente einer Menge nicht wie bei einem Suchbaum sortiert und durch Vergleiche wiedergefunden, sondern man berechnet mit Hilfe einer "Hashfunktion" aus dem Schlüssel einen Index (eine "Adresse"), unter dem der Schlüssel in einer "Hash"-Tabelle abgelegt wird.

In diesem Kapitel lernen Sie, welche Eigenschaften solch eine Hashfunktion besitzen muss, welche Funktionen in der Praxis eingesetzt werden, wie man durch Freihalten von Speicherplatz im Mittel eine konstante Such- und Einfügezeit erreicht, wie man das Löschen effizient behandelt und wie man den Speicherbereich dynamisch vergrößern kann. Zugleich werden Ihnen die hierfür benötigten Parameter (Tabellengröße, Auslastungsgrad, Kollisionsstrategien, Zyklenlänge) und ihre Bedeutung für Anwendungen vermittelt.

H.1 Einführung

Grunddidee: Schlüssel sollen durch einen einzigen Zugriff auf eine Tabelle (mit maximal p Einträgen) gefunden werden.

Gegeben sei eine Menge möglicher Schlüssel S und die Tabellengröße, dies ist eine natürliche Zahl $p << |S|$.

Es ist eine Abbildung $f: S \to \{0, 1, ..., p-1\}$ zu konstruieren, sodass es in einer zufällig ausgewählten n-elementigen Teilmenge $B = \{b_1, ..., b_n\} \subseteq S$ (mit $n \leq p$) im Mittel nur wenige Elemente $b_i \neq b_j$ mit $f(b_i) = f(b_j)$ gibt.

Die drei Operationen Suchen, Einfügen und Löschen müssen sehr "effizient" realisiert werden:

- Entscheide, ob $s \in S$ in B liegt (und gib an, wo). FIND
- Füge einen Schlüssel s in B ein. INSERT
- Entferne einen Schlüssel s aus B. DELETE

Für die Abbildung $f: S \rightarrow \{0, 1, ..., p\text{-}1\}$ müssen wir daher mindestens folgendes fordern:

- Sie muss surjektiv sein.
- Sie muss "gleichverteilt" sein, d.h., für jedes $0 \leq m < p$ sollte die Menge $S_m = \{s \in S \mid f(s) = m\}$ ungefähr $|S|/p$ Elemente enthalten.
- Sie muss schnell berechnet werden können.

Solch eine Abbildung f heißt Schlüsseltransformation oder **Hashfunktion**. (Genaueres siehe H.4.1.)

Diese Funktion verstreut die möglichen Schlüssel über den Indexbereich $\{0, 1, ..., p\text{-}1\}$. Daher der Name:

Hashing = (über eine Tabelle) gestreute Speicherung

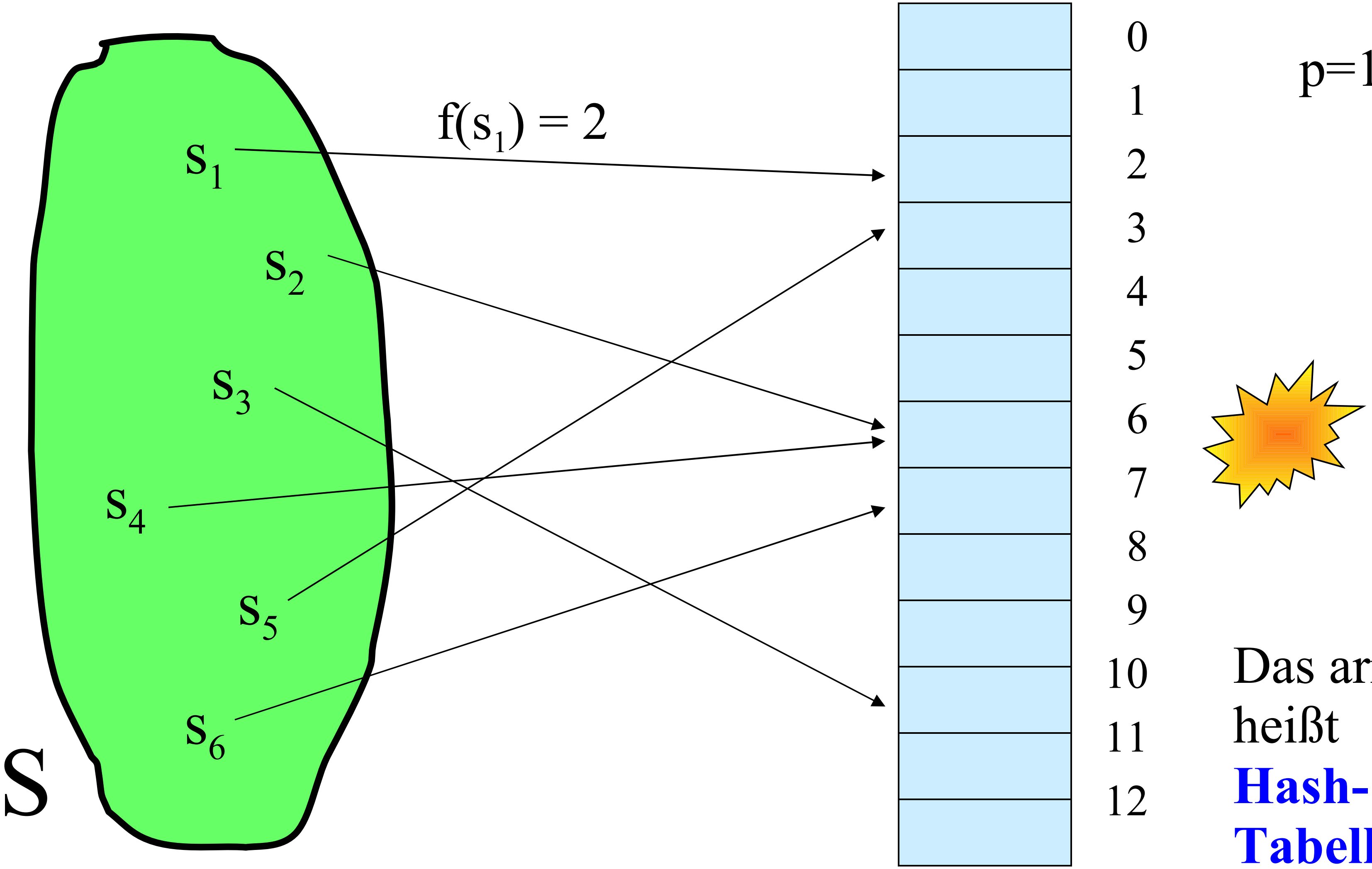
Nehmen wir an, wir hätten eine solche Abbildung $f: S \to \{0, 1, ..., p-1\}$, dann werden wir zur Speicherung von Teilmengen A von S ein Feld deklarieren:
A: array (0..p-1) of <Datentyp zur Menge S>

Jedes Element $s \in S$ wird unter der Adresse f(s) gespeichert, d.h., nach dem Speichern sollte $A(f(s)) = s$ gelten.

Um festzustellen, ob ein Schlüssel s in der jeweiligen Teilmenge liegt, braucht man nur zu prüfen, was in A(f(s)) steht. Doch es entstehen Probleme, wenn in der konkreten Teilmenge B mehrere Elemente mit gleichem f-Wert ("Kollisionen") 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 s_1 bis s_6 sollen gespeichert werden:


S
s_1
s_2
s_3
s_4
s_5
s_6
$f(s_1) = 2$
0
1
2
3
4
5
6
7
8
9
10
11
12

$p=13$

Das array heißt **Hash-Tabelle**.

Hier ist $f(s_2) = f(s_4) = 6$. Was nun?

Beispiel "modulo p"

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

Weiterhin sei p eine natürliche Zahl, $p>1$.

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

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

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, und als Menge B die Menge **A** der Monatsnamen:

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

Abzubildende Menge **A** :

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

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

für alle Wörter aus **A**. Zum Beispiel muss man für das Wort JANUAR als erstes die zu q gehörende Summe

$\varphi(J) = 10$ (im Falle $q=1$),

$\varphi(J) + \varphi(A) = 10 + 1 = 11$ (im Falle $q=2$),

$\varphi(J) + \varphi(A) + \varphi(N) = 10 + 1 + 14 = 25$ (im Falle $q=3$),

$\varphi(J) + \varphi(A) + \varphi(N) + \varphi(U) = 46$ (im Falle $q=4$) usw.

ermitteln. Wir listen zunächst diese Summen in der folgenden Tabelle für verschiedene q auf; hierbei kann man auch Buchstaben weglassen und H.B. nur den ersten und dritten Buchstaben betrachten ("`1.+3.`"); später müssen wir diese Werte modulo p (siehe übernächste Tabelle, die nur drei der sechs Spalten aus der anderen Tabelle benutzt) nehmen.

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

In der Tabellen haben wir bereits die genannte Modifikation für f benutzt, indem wir eine Teilmenge der Indizes $\{1,2,...,r\}$ ausgewählt und die zugehörigen Buchstabenwerte aufsummiert haben. 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 (= in ein array A = in eine "Hashtabelle" A) mit p Komponenten ein.

Für die Abbildung wählen wir willkürlich q=2 und p=22 und berechnen also hier die Hashfunktion

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

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. Die Menge $S = \Sigma^*$ ist unendlich; doch nehmen wir hier an, dass die tatsächlich benutzten Wörter der Menge S höchstens die Länge 20 haben (kürzere Wörter werden durch Zwischenräume, deren φ-Wert 0 sei, aufgefüllt) und deklarieren daher die Hashtabelle:

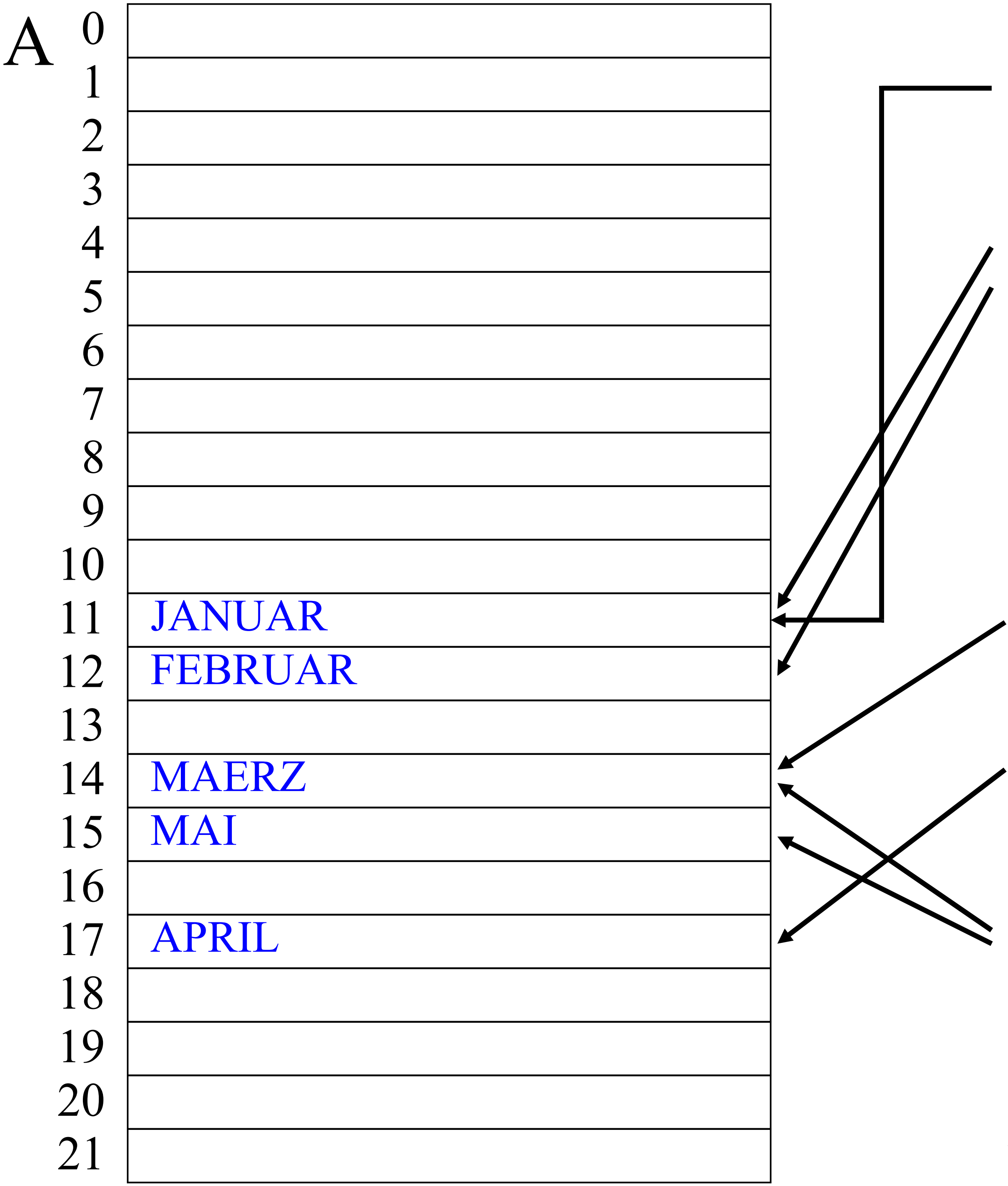
A: array (0..p-1) of String(20);

Es wird sicher auch folgender Fall auftreten:

Wir möchten einen Schlüssel s mit Hashwert f(s) = k in die array-Komponente A(k) eintragen; dort befindet sich jedoch bereits ein Schlüssel (es tritt ein "Konflikt" ein). Wir werden verschiedene Konfliktstrategien kennen lernen. Die einfachste ist sicherlich: Lege den Schlüssel s in die Komponente A(k+1); ist diese ebenfalls besetzt, so versuche es mit A(k+2) usw., wobei man von der Komponente A(p-1) nach A(0) übergeht (das Feld wird also als zyklisch aufgefasst).

Diese Vorgehensweise wird nun auf den nächsten Folien demonstriert; hierbei tritt bereits beim zweiten Schlüssel ein Konflikt auf.

Ein weiteres Beispiel finden Sie in Abschnitt H.6.



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

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

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

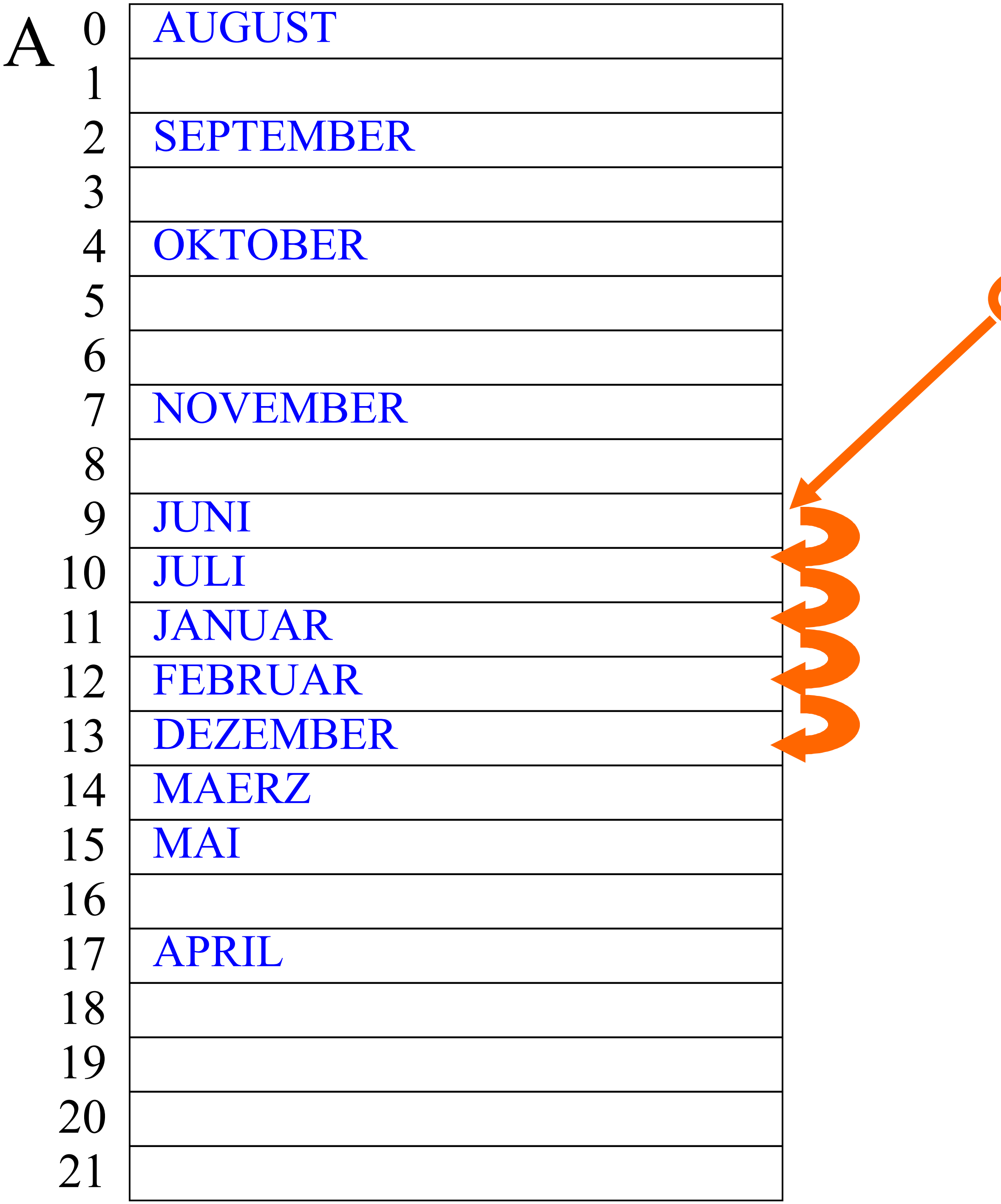
Konflikt! Verschiebe FEBRUAR um einen Platz nach hinten.

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.



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	

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 erneut **Konflikte** 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 Schlüssel.

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. Da A(9) besetzt ist, könnte JULI durch einen Konflikt verschoben worden sein, 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 JURA. Es ist f(JURA) = 9. Man prüft, ob A(9) = JURA ist. Dies trifft nicht zu, also geht man zu A(10) usw., bis man auf den leeren Eintrag A(16) trifft, dort bricht die Suche ab und JURA ist nicht in der Menge der Monatsnamen.

Wie löscht man? (Später!)

Wie viele Vergleiche braucht man, um einen Schlüssel 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 $19/12 \approx$ **1,6 Vergleiche**, falls der Schlüssel in der Menge ist (erfolgreiche Suche)

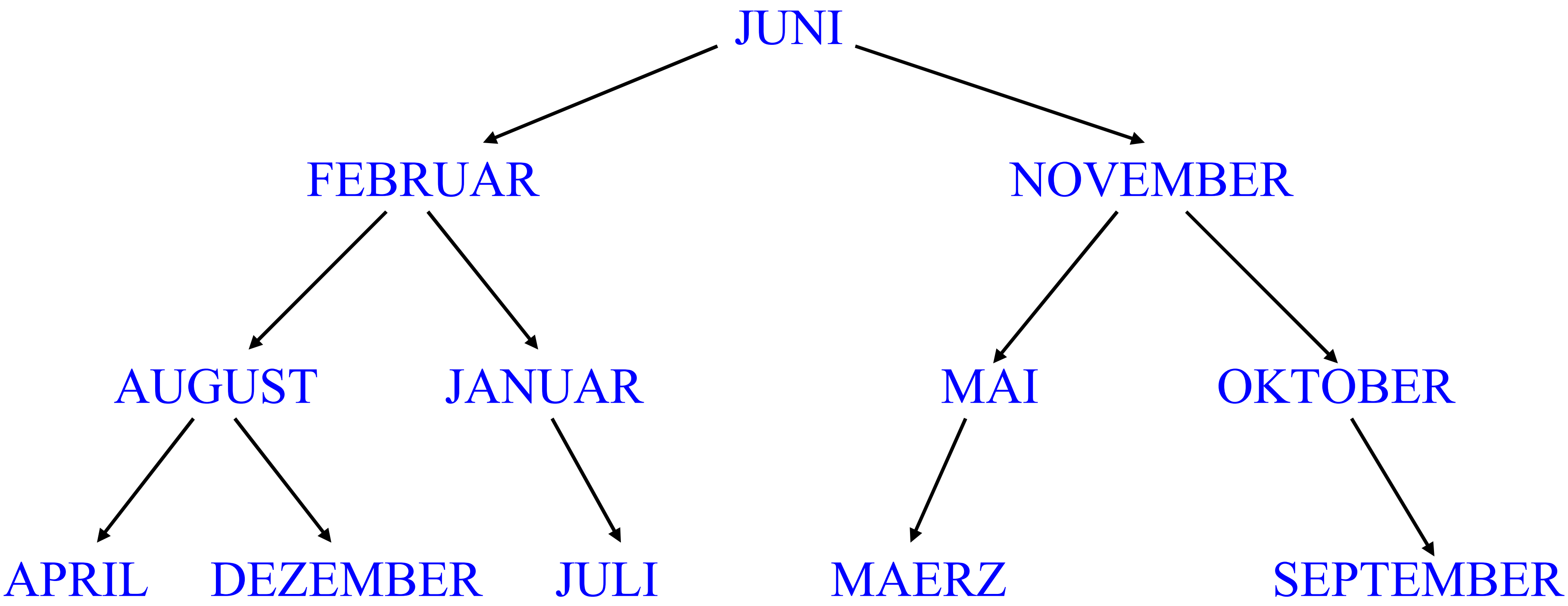
Wie viele Vergleiche braucht man für einen Schlüssel, der **nicht** in der Menge liegt? Gehe jede Komponente des Feldes hierzu durch (f soll gleichverteilt sein, siehe Anfang von Kapitel H.1):

0: 2 Vergleiche
1: 1 Vergleich
2: 2 Vergleiche
3: 1 Vergleich
4: 2 Vergleiche
5: 1 Vergleich
6: 1 Vergleich
7: 2 Vergleiche
8: 1 Vergleich
9: 8 Vergleiche
10: 7 Vergleiche
11: 6 Vergleiche
12: 5 Vergleiche
13: 4 Vergleiche
14: 3 Vergleiche
15: 2 Vergleiche
16: 1 Vergleich
17: 2 Vergleiche
18: 1 Vergleich
19: 1 Vergleich
20: 1 Vergleich
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 (erfolglose Suche).

Vergleich mit einem ausgeglichenen Suchbaum:



JUNI

FEBRUAR

NOVEMBER

AUGUST

JANUAR

MAI

OKTOBER

APRIL

DEZEMBER

JULI

MAERZ

SEPTEMBER

Mittlere Anzahl der Vergleiche für Elemente, die in der Menge sind: $(1+2+2+3+3+3+3+4+4+4+4+4)/12 \approx$ **3,1 Vergleiche**.
Falls das Element **nicht** in der Menge ist (13 null-Zeiger):
im Mittel $49/13 \approx$ **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 Schlüsselmenge S. Dafür sparen wir die Zeiger des Suchbaums. Es hängt also vom Platzbedarf ab, den jedes Element aus S 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.

H.2 Hashfunktionen

H.2.1 Aufgabe:
n Elemente einer (sehr großen) Menge S sollen in einem Feld array (0..p-1) of ... gesucht und dort in irgendeiner Reihenfolge eingefügt und gelöscht werden können.

Es sei |S| > p (sonst ist die Aufgabe ohne Konflikte durch irgendeine injektive Zuordnung zu lösen).

Benutze hierfür eine Funktion f: S → {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 das

H.2.2 Divisionsverfahren

1. Fasse den gegebenen Schlüssel s als Zahl auf.
2. Bilde den Rest der Division durch die Zahl p

$$f(s) = s \ \underline{\text{mod}}\ p.$$

Diese Restbildung hatten wir in H.1 benutzt.

(p wählt man meist als Primzahl, um Abhängigkeiten bei der Modulo-Bildung zu verringern; bei quadratischer Kollisionsstrategie, siehe unten, maximiert man hierdurch die Zykellänge.)

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

H.2.3 Multiplikationsverfahren:

(p ist die Größe der Hashtabelle)

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

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

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

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

Hinweise:

1. Die Zahl $|c_2| = 0.618\ 033\ 988\ 749\ 894\ 848\ 204 \ldots$ gilt als gut geeignete Zahl H.
2. Es kann auch $g(s) = \lceil s \cdot z \rceil - s \cdot z$ benutzt werden.

H.2.4: Wenn Zeichenfolgen als Schlüsselmenge $S = \Sigma^*$ vorliegen, wählt man gerne ein Teilfolgenverfahren (hier bzgl. der Division vorgestellt; analog: bzgl. der Multiplikation):

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

$$f(\alpha_1\alpha_2 \ldots\alpha_r) = \Big(\sum_{j=1}^{q} \varphi(\alpha_{i_j}) \Big) \underline{\text{mod}}\ p$$

oder verwende allgemein eine gewichtete Summe mit irgendwelchen speziell gewählten Zahlen $x_1, x_2, ..., x_q$:

$$f(\alpha_1\alpha_2 \ldots\alpha_r) = \Big(\sum_{j=1}^{q} x_j \cdot \varphi(\alpha_{i_j}) \Big) \underline{\text{mod}}\ p.$$

Definition H.2.5: Eine Hashfunktion $f: S \to \{0, 1, ..., p\text{-}1\}$ heißt perfekt *bzgl. einer Menge* $B \subseteq S$ (mit $|B| \leq p$) von Elementen, wenn f auf der Menge B injektiv ist, wenn also für alle Elemente $b_i \neq b_j$ aus B stets $f(b_i) \neq f(b_j)$ 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 solche perfekten Funktionen finden. Siehe Spalte `2.+3.` und $p = 17$ auf Folie 12.

H.3 Techniken beim Hashing

Man unterscheidet beim Hashing zwei Techniken: das geschlossene Hashing mit angefügten Überlaufbereichen und das offene Hashing, das alle Schlüssel im vorgegebenen Array unterbingt.

H.3.1 Geschlossenes Hashing

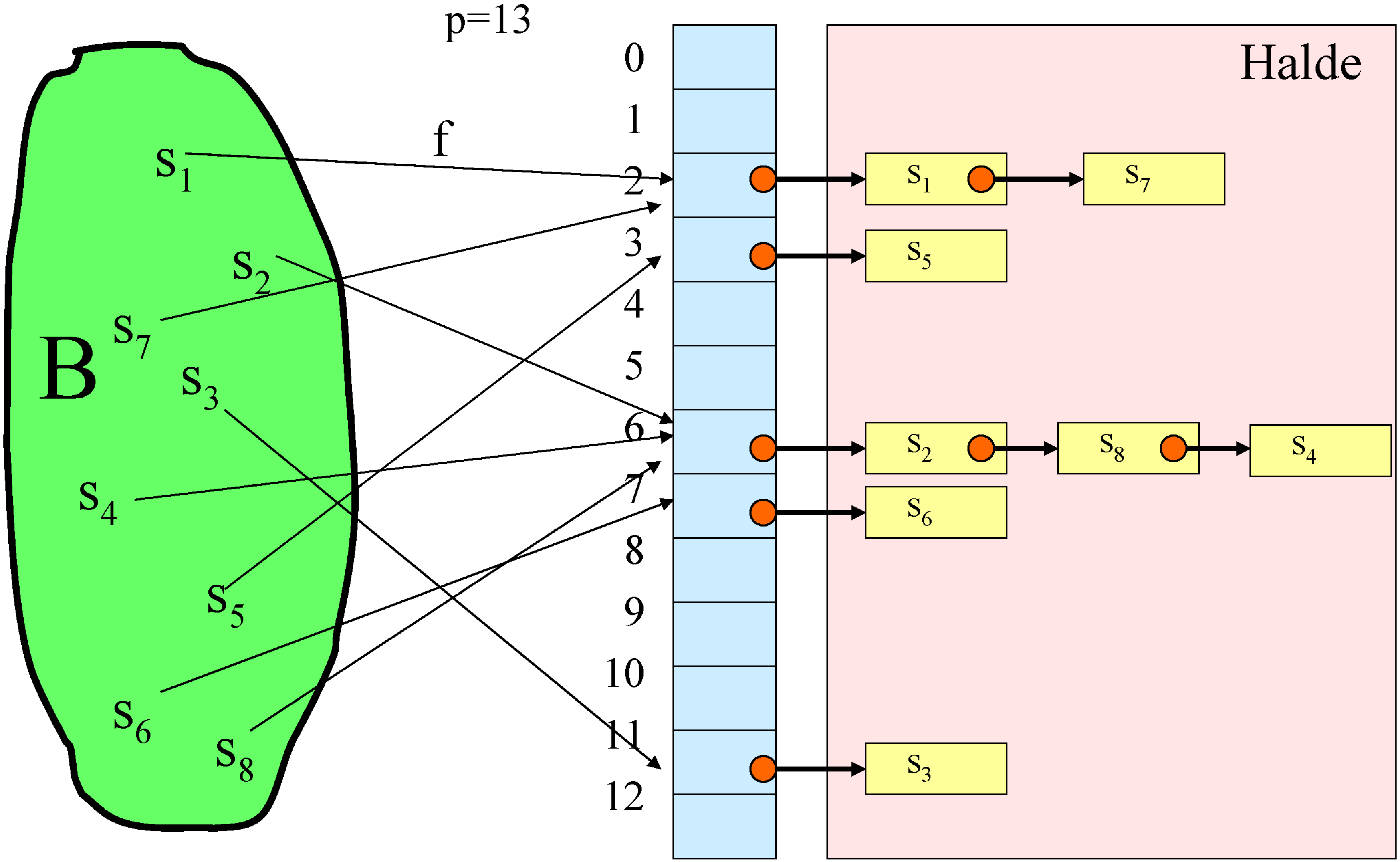
Beim *geschlossenen* Hashing lässt man keine Korrekturen des Hashwertes zu, sondern die Hash-Funktion führt zu einem Index, über den man zu einer Datenstruktur gelangt, an der sich der gesuchte Schlüssel befindet, befinden müsste oder an der er einzufügen ist. In der Regel werden alle gespeicherten Schlüssel, die den gleichen Hash-Wert besitzen, in eine lineare Liste oder in einen Suchbaum eingefügt.

Als Beispiel betrachten wir auf der nächsten Folie den Fall, dass alle Schlüssel, die den gleichen f-Wert haben, in einer linearen Liste gespeichert werden. In der Hashtabelle A steht an der Stelle A(i) der Zeiger auf die Liste der Schlüssel s mit $f(s)=i$.

Auf diese Weise entstehen keine Kollisionen in der Hashtabelle, sondern dieses Problem wird in die Verwaltung der p Listen verlagert.
(Hinweis: In der Praxis ist dies die "Haldenverwaltung".)

Die Hashtabelle ist somit nur eine Zugriffsstruktur für viele gleichartige Speicherstrukturen, die relativ klein gehalten werden können. Man spricht auch von externen Hashtabellen oder von Hashing mit externer Kollision.

Skizze: "Externe" Hashtabelle


p=13
f
B
s_1
s_2
s_7
s_3
s_4
s_5
s_6
s_8
0
1
2
3
4
5
6
7
8
9
10
11
12
Halde
s_1
s_7
s_5
s_2
s_8
s_4
s_6
s_3

Überlaufprobleme müssen mit der Haldenverwaltung gelöst werden!

Zeitaufwand beim geschlossenen Hashing: Das Suchen ("FIND") erfordert einen Zugriff auf das Feld A(f(s)) und anschließend muss die jeweilige Datenstruktur durchsucht werden.

Auch Einfügen ("INSERT") und Löschen ("DELETE") laufen bis auf den ersten Zugriff wie bei den zugrunde liegenden Datenstrukturen ab.

Vorteil des geschlossenen Hashings: Man muss keine feste obere Grenze für die Menge B der Schlüssel vorgeben. Vor allem, wenn viel gelöscht wird, kann man die schnellen Algorithmen der jeweiligen Datenstrukturen einsetzen. Insbesondere bei großen Datenbeständen lohnen sich Suchbäume.

Nachteil: Das Verfahren hängt von der Verwaltung der Listen (Zugriffszeiten, Garbage Collection) ab. (Offenes Hashing ist meist effizienter, siehe im Folgenden).

H.3.2 Offenes Hashing

Beim *offenen* Hashing wird der tatsächliche Index, unter dem der Schlüssel s später steht, erst mit dem Eintragen, also ggf. nach dem Durchlaufen einer Kollisionsstrategie, bestimmt. In diesem Fall befinden sich alle Schlüssel im Speicherbereich, den der Indexraum vorgibt (also im array (0..p-1)), und es gibt keine Überlaufbereiche.

Die Bezeichnungen „offen“ und „geschlossen“ sind anfangs verwirrend, da sie aus der Indexzuordnung abgeleitet wurden und sich nicht auf die Struktur des Speicherbereichs beziehen. Der Index bleibt also nach der Berechnung des Hashwertes "noch offen" und wird erst festgelegt, wenn ein freier Platz gefunden wurde.

Das Suchen geht in der Regel schnell, sofern mindestens 20% der Plätze des array frei gehalten werden.
Das Einfügen ist nicht schwierig.
Problem: Löschen.

H.3.3 Datentypen für das offene Hashing festlegen

(die Booleschen Werte brauchen wir erst später; in der Praxis speichert man den Hashwert f(s) des Schlüssels s zusätzlich ab; in der Regel lässt man die Komponente "Inhalt" weg, wenn der Schlüssel auf den eigentlichen Speicherort verweist):

```
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: ...

H.3.4 Einfüge-Algorithmus (für offenes Hashing)

```
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 s" loop
  if k < p then
       k := k+1; j := f(s);          -- j ist der Index in A für s
       if A(j).geloescht or not A(j).besetzt then  A(j).besetzt := true;
              A(j).Schluessel := s; 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;
```

H.3.5 Das Suchen erfolgt ähnlich:

Um einen Eintrag mit dem Schlüssel s zu finden, berechne f(s) und prüfe, ob in A(f(s)) ein Eintrag mit dem Schlüssel s steht. Falls ja, ist die Suche erfolgreich beendet, falls nein, prüfe A(f(s)).kollision. Ist dieser Wert false, dann ist die Suche erfolglos beendet, anderenfalls berechne mit der verwendeten Kollisionsstrategie (s.u.) einen neuen Platz j und prüfe erneut, ob der Schlüssel s gleich A(j).Schlüssel ist; falls ja, ist die Suche erfolgreich beendet, falls nein, prüfe den Wert von A(j).kollision. Ist dieser Wert false, dann ist die Suche erfolglos beendet, anderenfalls berechne mit der verwendeten Kollisionsstrategie einen neuen Platz j usw.

Wir wenden uns nun den Kollisionen und ihrer Behandlung zu.

Definition H.3.6: Sei f: S → {0, 1, ..., p-1} eine Hashfunktion.

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

Ist f(s') = 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 s' wird durch einen Schlüssel s" verursacht, der vom Hashwert her nicht an die Position i gehört und selbst durch Kollision hierhin 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 $f(\alpha_1\alpha_2 \ldots\alpha_r) = \left(\varphi(\alpha_1) + \varphi(\alpha_2) \right) \underline{\text{mod}}\ 14$. Drei Schlüssel werden in der Reihenfolge JANUAR, FEBRUAR, OKTOBER eingegeben. Es gilt:

f(JANUAR) = 11, f(FEBRUAR) = 11, f(OKTOBER) = 12.

Wir tragen JANUAR in der Komponente 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 nicht hingehört, sondern durch eine Kollision hierhin verschoben wurde. Folglich führt OKTOBER zu einer Sekundärkollision. (OKTOBER wird dann auf Platz A(13) eingetragen.)

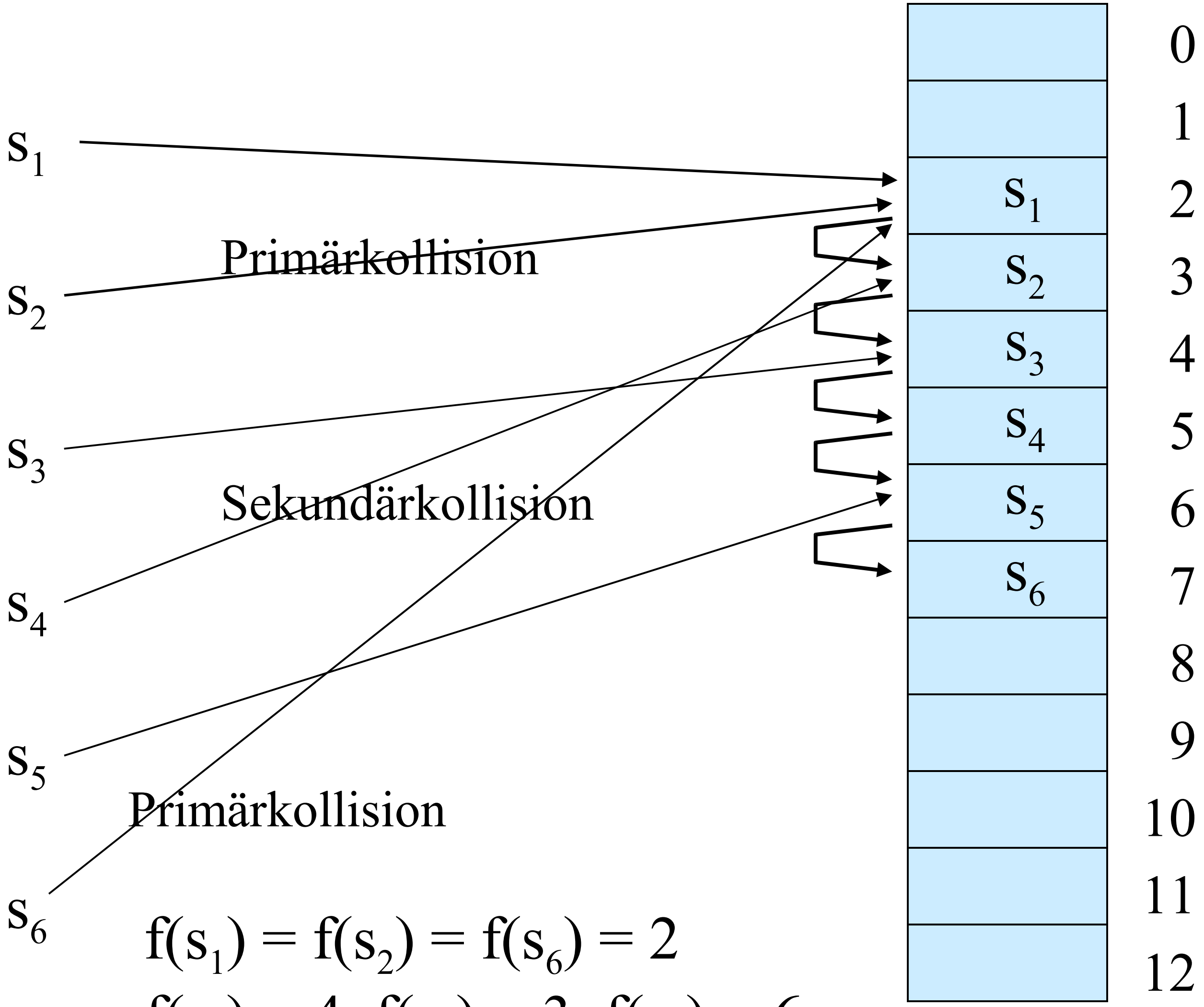
Definition H.3.7: Kollisionsstrategien ("Sondieren")

Der Schlüssel s soll stets unter dem Indes $f(s)$ gespeichert werden. Ist diese Komponente bereits besetzt, so versuche man, den Schlüssel s unter dem Index $G(s,i)$ einzufügen. Es sei i die Zahl der versuchten Zugriffe. c ist eine fest gewählte Konstante mit $ggT(c,p) = 1$; man kann stets $c=1$ wählen.

$G(s,i) = (f(s)+i \cdot c) \bmod p$	heißt "lineare Fortschaltung" oder "lineares Sondieren" oder "Lineares Hashing".
$G(s,i) = (f(s)+i^2) \bmod p$	heißt "quadratische Fortschaltung" oder "quadratisches Sondieren".

Beispielskizze: Lineares Sondieren mit c=1

p=13



s_1
s_2
s_3
s_4
s_5
s_6
Primärkollision
Sekundärkollision
Primärkollision
s_1
s_2
s_3
s_4
s_5
s_6
0
1
2
3
4
5
6
7
8
9
10
11
12

$f(s_1) = f(s_2) = f(s_6) = 2$

$f(s_3) = 4,\ f(s_4) = 3,\ f(s_5) = 6$

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 müssen die Kollisionketten immer nachvollzogen werden: Es bilden sich sog. Cluster, siehe Erläuterung unten.

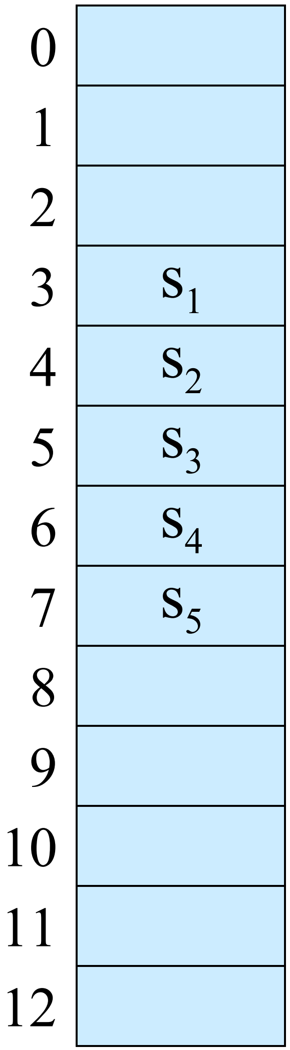
Der *Vorteil* des linearen Sondierens liegt darin, dass man bei dieser Kollisionsstrategie Werte aus der Hashtabelle wieder löschen kann.

Das quadratische Sondieren ist ebenfalls leicht zu implementieren. Sein *Vorteil* ist, dass die Sekundärkollisionen und damit die Clusterbildung reduziert 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, unzumutbar 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 H.4).

Um nicht in kurze Zyklen bei der Kollisionsstragegie zu gelangen, muss man beim quadratischen Sondieren unbedingt verlangen, dass die Größe der Hashtabelle p eine Primzahl ist. Dies wird in Satz H.3.10 begründet.

H.3.8 Clusterbildung bei linearem Sondieren



0	
1	
2	
3	s_1
4	s_2
5	s_3
6	s_4
7	s_5
8	
9	
10	
11	
12	

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

Es soll nun ein weiterer Schlüssel s eingefügt werden. Ist f(s) einer der Werte 3, 4, 5, 6 oder 7, so wird s 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 reduziert.

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 $f(\alpha_1\alpha_2 \ldots\alpha_r) = \left(2\varphi(\alpha_1) + \varphi(\alpha_2)\right) \underline{\text{mod}}\ 17.$

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 in die zunächst leere Tabelle ein. Es ergibt sich am Ende die nebenstehende Tabelle.

H.3.9: Länge von Zyklen bei Kollisionsstrategien

Wir müssen uns nun überzeugen, dass bei den Kollisionsstrategien keine zu kurzen 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 $(f(s)+i \cdot c)$ mod p (für $i = 0, 1, 2, \ldots$) 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 nehmen hier an, dass $p > 2$ und somit ungerade ist.)

Wir fragen daher: Wann tritt in der Folge der Zahlen

$(f(s)+i^2)$ mod p 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

$(f(s)+i^2)\ \underline{\text{mod}}\ p\ =\ (f(s)+j^2)\ \underline{\text{mod}}\ p$,

d.h. $(i^2-j^2)\ \underline{\text{mod}}\ p = (i+j)\cdot(i-j)\ \underline{\text{mod}}\ p = 0$.

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 H.3.10 (Zykluslänge beim quadratischen Sondieren)
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 $j = (p-1)/2$ ist $(i+j)\cdot(i-j)\ \underline{\text{mod}}\ p = 0$, da $i+j=p$ und $i-j=1$ gilt. Die in Satz H.3.10 genannte Länge eines Zyklus von $(p+1)/2$ Schritten tritt somit für *alle* Zahlen (auch für die Primzahlen) auf.
Hinweis: Bei einer Primzahl p kommen die "komplementären Zahlen" $(-i^2)\ \underline{\text{mod}}\ p$ beim quadratischen Sondieren nicht vor. Modifiziert man daher das quadratische Sondieren so, dass zwischen $i^2\ \underline{\text{mod}}\ p$ und $(i+1)^2\ \underline{\text{mod}}\ p$ immer $(-i^2)\ \underline{\text{mod}}\ p$ eingeschoben wird, so erreicht man die volle Zykluslänge p.

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

H.3.11 Löschen in Hashtabellen (DELETE)

Dieses bildet das Hauptproblem beim offenen Hashing.

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

Nachteil: Wenn oft gelöscht wird, dann ist die Tabelle schnell voll und muss mit gewissem Aufwand reorganisiert werden, vgl. Abschnitt H.4. Dennoch ist dieses Vorgehen in der Praxis gut einsetzbar.

H.4 Rehashing

Was muss man tun, wenn der Auslastungsgrad über 80% hinausgeht oder gar den Wert 1 erreicht? Dann muss man die Hashtabelle verlängern, also p durch eine Zahl p' > p ersetzen, hierfür eine neue Hashfunktion festlegen und die neue Hashtabelle aus der alten Tabelle, in der die Schlüssel in den Plätzen von 0 bis p-1 standen, errechnen.

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 und eine Umorganisation mit dem gleichen p notwendig wird (Rehashing mit p=p').

H.4.1 Beispiel für p = 7 und p' = 13

	0
	1
MAE	2
JAN	3
FEB	4
APR	5
MAI	6

p=7

Wörter:
JAN, FEB, MAE, APR, MAI.

Alte Hashfunktion (lin. Sond.):

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

0	
1	
2	
3	
4	
5	MAE
6	APR
7	
8	FEB
9	MAI
10	
11	JAN
12	

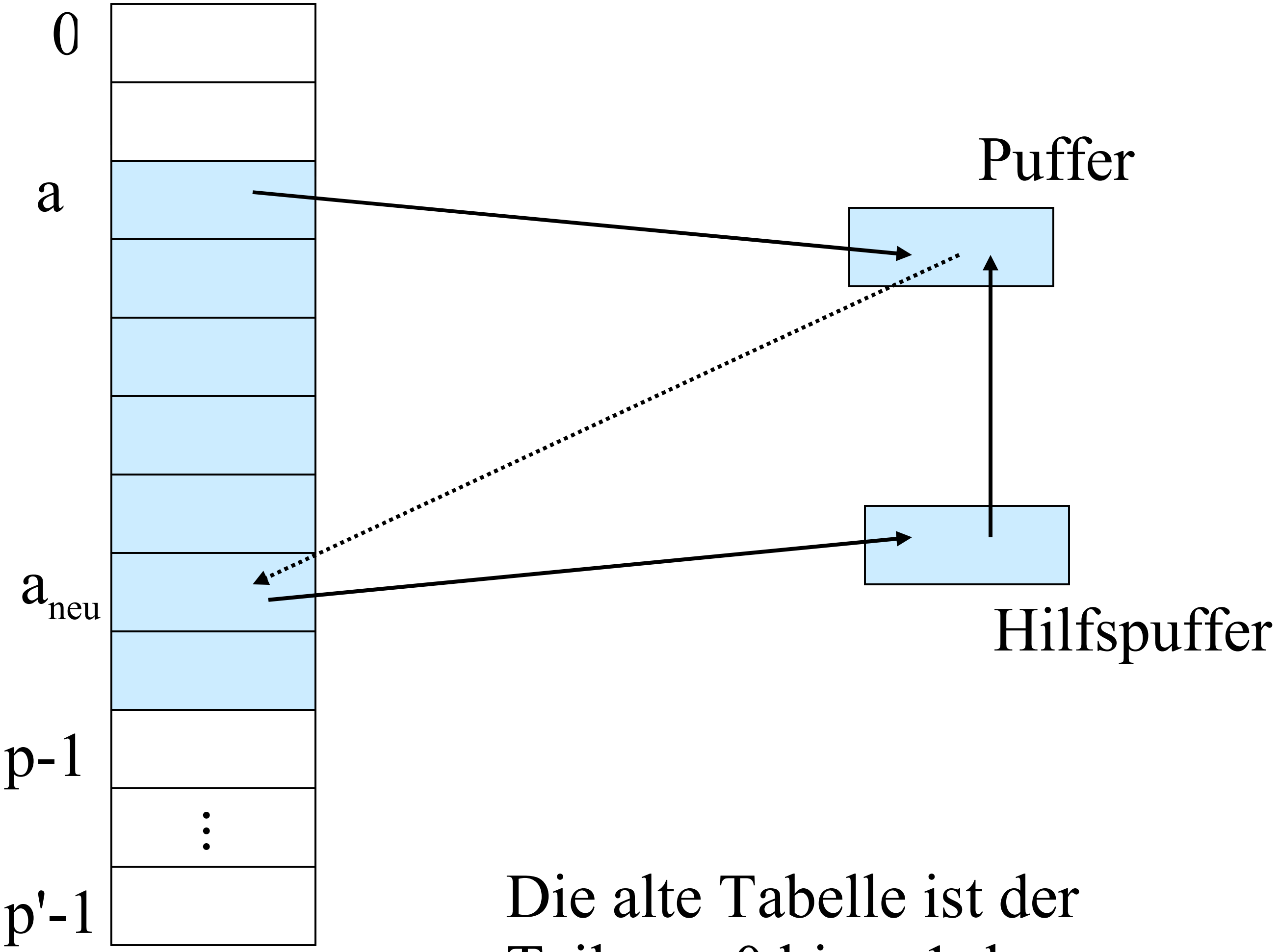
p'=13

Alle Wörter müssen übertragen werden.

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

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

H.4.2 Rehashing: Durchlaufe die neue Tabelle von 0 bis p-1:


0
a
a_{neu}
p-1
⋮
p'-1
Puffer
Hilfspuffer

Neue Tabelle
mit p' Plätzen

Die alte Tabelle ist der Teil von 0 bis p-1 der neuen Tabelle.