2 Streuwerttabellen (hash tables)

2.1 Einleitung und Motivation

2.1.1 Aufgabe

- Schreiben Sie ein Programm (z. B. in C, C++ oder Java),
  - das als Eingabe eine Folge von Wörtern erhält (z. B. ein Wort pro Zeile)
  - und als Ausgabe die Häufigkeit jedes Worts liefert.

- Beispieleingabe (links) und zugehörige Ausgabe
  (rechts, die Reihenfolge der Ausgabezeilen ist beliebig):

```
eins
zwei
drei
zwei
eins
eins
```

```
3 eins
2 zwei
1 drei
```

2.1.2 Lösung mit verketteter Liste

```
#include <iostream>      // cin, cout, endl, getline
#include <string>        // string
using namespace std;

using uint = unsigned int; // Vorzeichenlose ganze Zahl.

// Listenelement.
struct Elem {
  string word;           // Wort.
  uint count;            // Häufigkeit.
  Elem* next;            // Verkettung.

  // Initialisierung mit Wort w, Häufigkeit 1 und Verkettung n.
  Elem (string w, Elem* n) : word(w), count(1), next(n) {}
};

// Hauptprogramm.
int main () {
  Elem* h = nullptr;     // Listenanfang (head).
  string w;              // Aktuelles Wort.
```

```
  // Eingabe zeilenweise lesen und verarbeiten.
  while (getline(cin, w) && w != "") {
    // Liste nach einem Element e mit Wort w durchsuchen.
    Elem* e;
    for (e = h; e != nullptr; e = e->next) {
      // Wenn gefunden, Zähler erhöhen und abbrechen.
      if (e->word == w) {
          e->count++;
          break;
      }
    }
    // Andernfalls neues Element mit Zähler 1
    // am Anfang der Liste einfügen.
    if (e == nullptr) h = new Elem(w, h);
  }

  // Liste durchlaufen und Ergebnisse ausgeben.
  for (Elem* e = h; e != nullptr; e = e->next) {
    cout << e->count << " " << e->word << endl;
  }
}
```

- Problem: Linearer Aufwand für jede Suche

2.1.3 Lösung mit Suchbaum

- Zum Beispiel AVL-Baum oder Rot-schwarz-Baum
- Siehe „Algorithmen und Datenstrukturen 1“
- Probleme:
  - Immer noch logarithmischer Aufwand für jede Suche
  - Jeder Baumknoten braucht zusätzlich Platz für Zeiger und ggf. Farbinformation

2.1.4 Lösung mit Streuwerttabelle (noch fehlerhaft)

```
#include <iostream>      // cin, cout, endl, getline
#include <string>        // string
using namespace std;

using uint = unsigned int; // Vorzeichenlose ganze Zahl.

// Streuwert der Zeichenfolge s ermitteln.
uint hashval (string s) {
  // Jedes Zeichen s[i] von s entspricht irgendeinem Zahlenwert.
  // Diese Werte werden relativ willkürlich zu einem Gesamtwert
  // verknüpft.
  // Überlauf ist bei vorzeichenloser Arithmetik wohldefiniert.
  uint h = 0;
  for (uint i = 0; i < s.length(); i++) {
    h = h * 31 + s[i];
  }
  return h;
}

// Tabellengröße.
const int N = 100;
```

```
// Tabellenelement.
struct Elem {
  string word;            // Wort.
  uint count;             // Häufigkeit.

  // Initialisierung mit leerem Wort und Häufigkeit 0.
  // Dieser Konstruktor wird gebraucht, weil er automatisch
  // für alle Elemente einer Reihe aufgerufen wird.
  Elem () : word(""), count(0) {}

  // Initialisierung mit Wort w und Häufigkeit 1.
  Elem (string w) : word(w), count(1) {}
};

// Hauptprogramm.
int main () {
  Elem tab [N];           // Tabelle.
  string w;               // Aktuelles Wort.

  // Eingabe zeilenweise lesen und verarbeiten.
  while (getline(cin, w) && w != "") {
    // Streuwert des Worts als Index in die Tabelle verwenden.
    uint i = hashval(w) % N;
```

```
      // Wenn dort bereits ein Element ist, Zähler erhöhen.
      // FEHLER: Wenn dort ein anderes Wort als w steht, wird
      // dessen Zähler fälschlich erhöht, und w "geht verloren".
      if (tab[i].count > 0) {
        tab[i].count++;
      }
      // Andernfalls neues Element mit Zähler 1 eintragen.
      else {
        tab[i] = Elem(w);
      }
    }

    // Tabelle durchlaufen und Ergebnisse ausgeben.
    for (int i = 0; i < N; i++) {
      if (tab[i].count > 0) {
        cout << tab[i].count << " " << tab[i].word << endl;
      }
    }
  }
```

Diskussion

- Vorteile
  - Konstanter Aufwand für jede „Suche“ → maximal effizient
  - Kompakte Speicherung der Tabelle als Reihe
- Nachteil
  - Tabelle kann nicht wachsen (müsste bei Bedarf umkopiert werden)
- Problem
  - Verschiedene Wörter können „zufällig“ den gleichen Streuwert besitzen (konkret z. B. `Aa` und `BB`).
  - Derartige Wörter werden noch nicht korrekt gezählt.
- Fragen
  - Wie löst man dieses Problem?
  - Wie berechnet man grundsätzlich sinnvolle Streuwerte?

2.1.5 Lösung mit `std::unordered_map`

```
#include <iostream>      // cin, cout, endl, getline
#include <string>        // string
#include <unordered_map>// unordered_map
#include <utility>       // pair
using namespace std;

// Hauptprogramm.
int main () {
  unordered_map<string, unsigned int> tab (100); // Tabelle.
  string w;              // Aktuelles Wort.

  // Eingabe zeilenweise lesen und verarbeiten.
  while (getline(cin, w) && w != "") {
    // tab[w] liefert den Zähler zu Wort w mit Anfangswert 0.
    tab[w]++;
  }
  // Ausgabe produzieren.
  for (pair<string, int> e : tab) {
    cout << e.second << " " << e.first << endl;
  }
}
```

2.1.6 Lösung in der Skriptsprache AWK

```
# Eingabe zeilenweise lesen und verarbeiten.
{ tab[$0]++ }

# Ausgabe produzieren.
END { for (w in tab) print tab[w], w }
```

Erläuterungen

- Der Block `{ ..... }` wird automatisch für jede Eingabezeile ausgeführt. Der Inhalt der Zeile ist jeweils als `$0` verfügbar.
- Die Reihe `tab` ist in Wirklichkeit ein assoziativer Container analog zu `java.util.Map` und `std::unordered_map`, der intern (vermutlich) als Streuwerttabelle implementiert ist.
- Der Block `END { ..... }` wird automatisch nach Verarbeitung der Eingabe ausgeführt.
- Die Schleife `for (w in tab) .....` durchläuft alle Schlüsselwerte der Tabelle `tab`.

2.1.7 Allgemeines Prinzip

- Die in einer Streuwerttabelle gespeicherten Objekte bestehen normalerweise aus einem *Schlüssel* (im Beispiel ein Wort) und einem zugehörigen *Wert* (im Beispiel der zugehörige Zähler).

- Schlüssel und Wert können selbst wiederum aus mehreren Teilen bestehen.
  In einer Tabelle mit Personendaten könnte der Schlüssel eines Objekts z. B. aus Vor- und Nachname einer Person bestehen, während der Wert aus beliebig vielen weiteren Daten bestehen kann (z. B. Geburtsdatum, Adresse etc.).

- Die Werte können aber auch fehlen, z. B. wenn das Beispielprogramm nur die Menge aller verschiedenen Wörter ohne ihre Häufigkeit ausgeben soll.
  (Dann bezeichnet man die Streuwerttabelle eher als *Streuwertmenge*.)

- Für die *Gleichheit* zweier Objekte sind in jedem Fall nur ihre Schlüssel relevant, ebenso für die Berechnung ihres Streuwerts.

- Gleiche Objekte müssen immer den gleichen Streuwert besitzen.

- Umgekehrt kann es aber durchaus auch verschiedene Objekte mit dem gleichen Streuwert geben.

2.1.8 Grundoperationen auf Streuwerttabellen

- Einfügen/Ersetzen eines Objekts (k, v) mit Schlüssel k und Wert v put(k, v)
  1. Wenn die Tabelle bereits ein Objekt (k', v') mit $k' = k$ und irgendeinem Wert v' enthält, wird es durch (k, v) ersetzt.
  2. Andernfalls wird das Objekt (k, v) zur Tabelle hinzugefügt, sofern diese noch nicht voll ist.
- Suchen eines Schlüssels k get(k)
  1. Wenn die Tabelle ein Objekt (k', v') mit $k' = k$ und irgendeinem Wert v' enthält, wird der Wert v' zurückgeliefert.
  2. Andernfalls wird das Nichtvorhandensein eines solchen Objekts durch einen geeigneten Resultatwert $\perp$ (nil, d. h. logisch nichts) angezeigt.
- Löschen eines Schlüssels k remove(k)
  1. Wenn die Tabelle ein Objekt (k', v') mit $k' = k$ und irgendeinem Wert v' enthält, wird es aus der Tabelle entfernt.
  2. Andernfalls ist die Operation wirkungslos.

Die Operationen sind hier bewusst abstrakt, d. h. unabhängig von einer konkreten Implementierung definiert. In § 2.5.1 und § 2.6.1 werden sie dann jeweils für eine bestimmte Implementierung konkretisiert.

2.1.9 Weitere Anwendungsbeispiele für Streuwerttabellen

- Online-Katalog → Bestellnummer als Schlüssel
- Personaldatenbank → Personalnummer als Schlüssel
- Kfz-Datenbank → Kennzeichen als Schlüssel
- Cache eines Webbrowsers → URL als Schlüssel
- Variablentabelle eines Compilers → Variablenname als Schlüssel
- Große, dünn besetzte Matrix → Paar von Zeilen- und Spaltenindex als Schlüssel

2.2 Streuwertfunktionen

2.2.1 Grundregeln

- Eine *Streuwertfunktion* ordnet jedem Objekt (aus einer bestimmten Grundmenge) bzw. jedem Schlüssel eine (vorzeichenlose) ganze Zahl zu.
- Inhaltlich gleiche Objekte *müssen* den gleichen Streuwert besitzen (vgl. § 2.1.7).
- Inhaltlich verschiedene Objekte *sollten* möglichst verschiedene Streuwerte besitzen.
- Letzteres ist aber häufig nicht möglich, weil es z. B. viel mehr (prinzipiell unendlich viele) inhaltlich verschiedene `string`-Objekte als `uint`-Werte gibt.

2.2.2 Beispiel: Zeichenketten

- Siehe § 2.1.4.

2.2.3 Beispiel: Punkte

```
// Punkt im zweidimensionalen Raum.
struct Point {
  // Koordinaten des Punkts.
  int x, y;
};

// Sind die Punkte p1 und p2 inhaltlich gleich?
// Durch diese Definition von operator== können Point-Objekte
// mit dem normalen Gleichheitsoperator (==) verglichen werden.
bool operator== (Point p1, Point p2) {
  // Hier wird zweimal der Gleichheitsoperator für int verwendet.
  return p1.x == p2.x && p1.y == p2.y;
}

// Der Ungleichheitsoperator (!=) muss aber bei Bedarf zusätzlich
// definiert werden.
bool operator!= (Point p1, Point p2) {
  // Hier wird der zuvor definierte Gleichheitsoperator für Point
  // verwendet.
  return !(p1 == p2);
}
```

```
// Vorzeichenlose ganze Zahl.
using uint = unsigned int;

// Streuwert des Punkts p berechnen und zurückliefern.
uint hashval (Point p) {
  // Einfache Möglichkeit:
  // Summe der Koordinaten von p liefern (die dann implizit
  // von int nach uint umgewandelt wird).
  // Dann haben aber Punkte (a, b) und (b, a) immer den gleichen
  // Streuwert, obwohl sie für a != b inhaltlich verschieden sind.
  // return p.x + p.y;

  // Bessere Möglichkeit:
  // Beide Koordinaten als Bitmuster interpretieren;
  // Bitmuster von x um 16 Positionen nach links verschieben und
  // dann mit dem Bitmuster von y durch bitweises Oder verknüpfen.
  // Hier haben Punkte (a, b) und (b, a) für viele a != b
  // verschiedene Streuwerte.
  // (Aber natürlich können auch hier inhaltlich verschiedene
  // Punkte immer noch den gleichen Streuwert besitzen.)
  return p.x << 16 | p.y;
}
```

2.3 Einschränkung des Wertebereichs einer Streuwertfunktion

2.3.1 Grundsätzlich

- Damit ein Streuwert `h` als Index `i` in eine Tabelle (array) der Größe `N` verwendet werden kann, muss er noch auf den Wertebereich von `0` einschließlich bis `N` ausschließlich eingeschränkt werden.

2.3.2 Divisionsrestmethode

- Mathematisch: `i = h mod N`
- Aber Achtung: In C, C++, Java und vielen anderen Programmiersprachen stimmt `h % N` nur für `h >= 0` (und `N > 0`) garantiert mit der mathematischen Definition von `h mod N` überein.
- Beispielsweise ist `-14 % 3` meist gleich `-2` (statt `1`) und liegt damit nicht im verlangten Wertebereich.
- Bei Verwendung vorzeichenloser Werte kann dieses Problem nicht auftreten.
- Andere mögliche Lösungen:

```
i = (h >= 0 ? h : -h) % N
i = (h % N + N) % N
```

- Erfahrungsgemäß hängt die Güte der Divisionsrestmethode stark vom Wert `N` ab:
  - Wenn `N` eine Zweierpotenz 2^p ist, besteht `h mod N` aus den niedrigsten p Bits des Werts `h`, d. h. die übrigen Bits werden ignoriert, was zu schlechter Streuung führen kann.
  - Gut geeignet sind meist Primzahlen `N`, die nicht zu nahe an einer Zweierpotenz liegen.

2.3.3 Multiplikationsmethode

Mathematisch

- Gegeben sei eine beliebige Konstante $A \in (0, 1)$,
  z. B. $A = \frac{\sqrt{5}-1}{2} = 0.61803399\ldots$ (vgl. Goldener Schnitt).

- Berechne:

  $$u = h \cdot A,$$
  $$v = u \bmod 1 = u - \lfloor u \rfloor \in [0, 1),$$
  $$i = \lfloor v \cdot N \rfloor \in \{0, \ldots, N-1\},$$

  d. h. multipliziere die Nachkommastellen von $h \cdot A$ mit N.

- Erfahrungsgemäß funktioniert diese Methode mit den meisten Werten von N und A gut.

- Laut Knuth liefert die obige Wahl von A aber besonders gute Ergebnisse.

- Für $A = \frac{1}{N}$ erhält man gerade die Divisionsrestmethode.

Praktische Berechnung mit Ganzzahlarithmetik

- Die Tabellengröße N muss hierfür eine Zweierpotenz 2^p sein.
- Die Wortbreite des Prozessors sei w, d. h. ganze Zahlen bestehen aus w Bits.
- Gegeben sei eine beliebige ganzzahlige Konstante $A' \in \left(0, 2^w\right)$ anstelle von $A \in (0, 1)$, aus der $A = \frac{A'}{2^w}$ berechnet werden könnte.
- Berechne ganzzahlig mit Wortbreite w:

$$u' = h \cdot A',$$

$$v' = u' \bmod 2^w,$$

$$i' = v' \cdot N = v' \cdot 2^p,$$

$$i = \frac{i'}{2^w} = \frac{v' \cdot 2^p}{2^w} = \frac{v'}{2^{w-p}} \in \{0, \ldots, N-1\}.$$

- Damit sind A', u', v' und i' jeweils um den Faktor 2^w größer als A, u, v bzw. i.

- $h \cdot A'$ ist prinzipiell eine Zahl mit $2w$ Bits, von der die oberen w Bits bei der praktischen Berechnung mit Wortbreite w jedoch verlorengehen, sodass das Ergebnis dieser Berechnung tatsächlich bereits v' entspricht.

- Die abschließende Division durch die Zweierpotenz 2^{w-p} kann effizient durch eine Bitverschiebung um $w - p$ Positionen nach rechts ausgeführt werden.

- Damit verwendet man faktisch die obersten p Bits des Produkts $h \cdot A'$ als Index.

- Konkret z. B. in C++ mit `s` = A':

```
i = (h * s) >> (sizeof(h * s) * CHAR_BIT - p)
```

  Der Typ von `h * s` muss hierfür vorzeichenlos sein (was automatisch der Fall ist, wenn der Typ von `h` vorzeichenlos ist), damit das Resultat der Bitverschiebung garantiert der gewünschten Division entspricht.

- `CHAR_BIT` ist in `<climits>` definiert und bezeichnet die Anzahl der Bits eines `char`-Werts, d. h. eines Bytes (die prinzipiell auch größer als 8 sein könnte).
  Damit entspricht `sizeof(h * s) * CHAR_BIT` genau der Wortbreite w von `h * s`.

Beispiel

- Sei $N = 2^{10} = 1024$ und $A = \frac{\sqrt{5}-1}{2} = 0.61803399\ldots$
- Dann ergibt sich für den Streuwert $h = 15341$ mit Gleitkomma-Arithmetik:

  $u = h \cdot A = 9481.25942141\ldots$

  $v = u \bmod 1 = 0.25942141\ldots$

  $i = \lfloor v \cdot N \rfloor = 265$

- Mit 16-Bit-Ganzzahlarithmetik und $A' = A \cdot 2^{16} \approx 40503$ ergibt sich:

  $u' = h \cdot A' = 621356523 = 0010010100001001\ 0010010111101011_2$

  $v' = u' \bmod 2^{16} = 9707 = 0010010111101011_2$ (die hinteren 16 Bit von u')

  $i = \frac{v'}{2^{16-10}} = 151 = 0010010111_2$ (die obersten 10 Bit von v')

- Aufgrund von Rundungsfehlern ergeben sich also unterschiedliche Ergebnisse, was für die Praxis jedoch kein Problem darstellt.

2.4 Behandlung von Kollisionen

- Wenn zwei verschiedene Objekte bzw. Schlüssel (nach der Einschränkung des Wertebereichs) den gleichen Streuwert bzw. Index besitzen, spricht man von einer *Kollision*.

- Zur Auflösung solcher Kollisionen gibt es prinzipiell zwei Möglichkeiten:
  - *Verkettung:*
    Alle Objekte mit dem gleichen Streuwert bzw. Index werden in einer verketteten Liste am gleichen Platz der Tabelle gespeichert.
  - *Offene Adressierung:*
    Wenn der Platz, in dem ein Objekt eigentlich gespeichert werden sollte, bereits belegt ist, wird nach irgendeiner geeigneten Methode ein noch freier „Ersatzplatz“ gesucht.

- Beide Verfahren mit ihren unterschiedlichen Vor- und Nachteilen werden im folgenden genauer betrachtet.

2.5 Verkettung (chaining)

2.5.1 Grundoperationen

- Gegeben sei eine Reihe *tab* der Größe N zur Speicherung verketteter Listen sowie eine Funktion h, die jedem Schlüssel k einen Index $i \in \{0, \ldots, N-1\}$ zuordnet. (Das heißt, h ist eine Streuwertfunktion mit Einschränkung des Wertebereichs.)

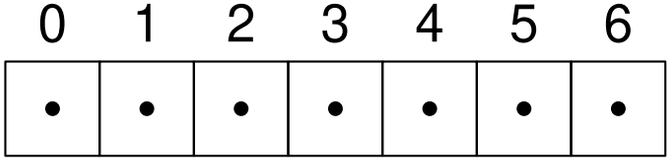
- Einfügen/Ersetzen eines Objekts (k, v) mit Schlüssel k und Wert v
  1. Berechne den Index $i = h(k)$ und durchsuche die Liste $tab[i]$ nach einem Objekt (k', v') mit $k' = k$ und irgendeinem Wert v'.
  2. Wenn es ein solches Objekt gibt, ersetze es durch das Objekt (k, v).
  3. Andernfalls füge das Objekt (k, v) am Anfang der Liste ein. (Prinzipiell könnte das Objekt irgendwo in die Liste eingefügt werden, aber am Anfang geht es am einfachsten und schnellsten.)

- Suchen eines Schlüssels k
  1. Berechne den Index $i = h(k)$ und durchsuche die Liste $tab[i]$ nach einem Objekt (k', v') mit $k' = k$ und irgendeinem Wert v'.
  2. Wenn es ein solches Objekt gibt, liefere den Wert v' zurück.
  3. Andernfalls liefere $\perp$.

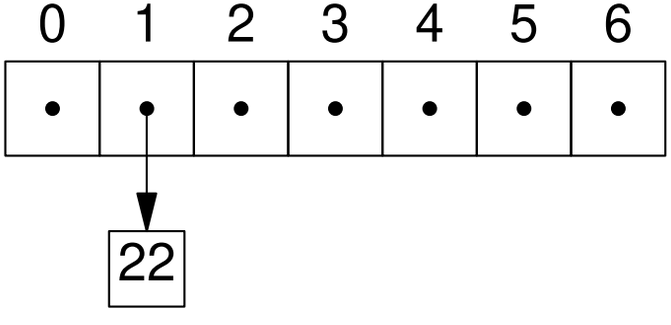
- Löschen eines Schlüssels k
  1. Berechne den Index $i = h(k)$ und durchsuche die Liste $tab[i]$ nach einem Objekt (k', v') mit $k' = k$ und irgendeinem Wert v'.
  2. Wenn es ein solches Objekt gibt, entferne es aus der Liste.

2.5.2 Beispiel

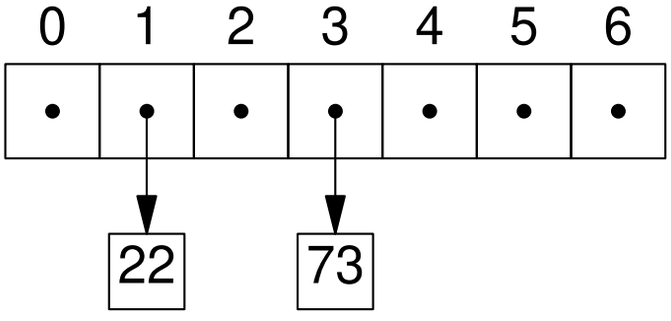
- Sei $N = 7$ und $h(k) = k \bmod 7$ (d. h. die Schlüssel sind bereits ganze Zahlen).
- Leere Tabelle:



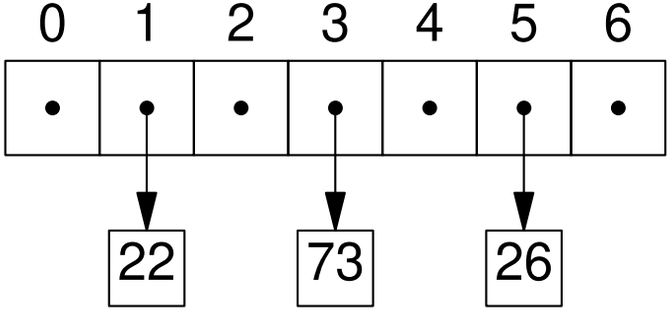
- Einfügen von $k = 22$ mit $h(k) = 22 \bmod 7 = 1$:



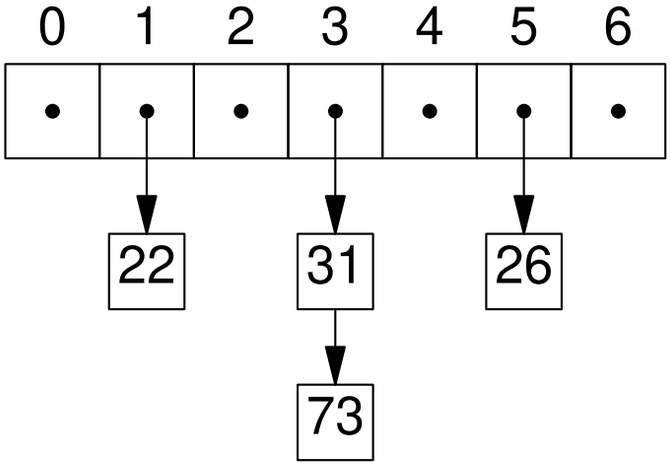
- Einfügen von $k = 73$ mit $h(k) = 73 \bmod 7 = 3$:



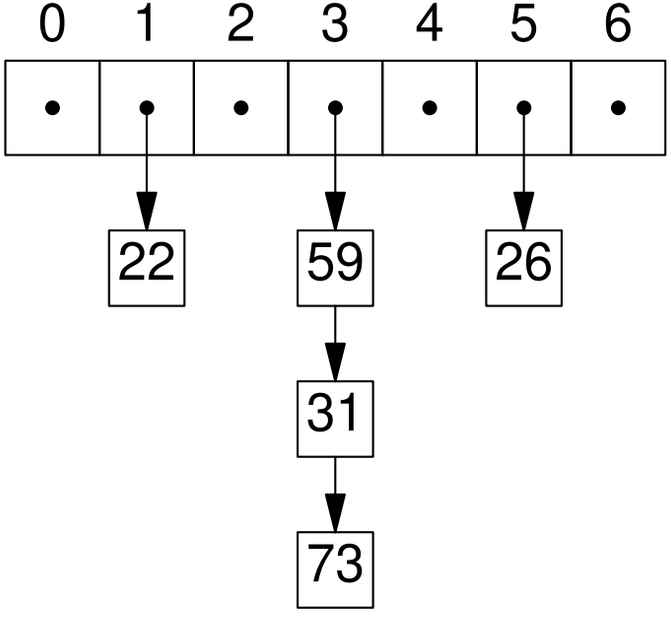
- Einfügen von $k = 26$ mit $h(k) = 26 \bmod 7 = 5$:



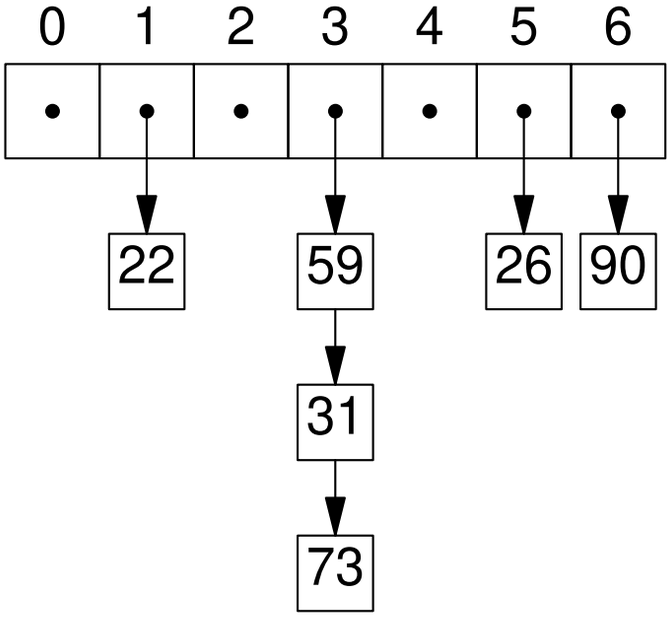
❒ Einfügen von $k = 31$ mit $h(k) = 31 \bmod 7 = 3$:



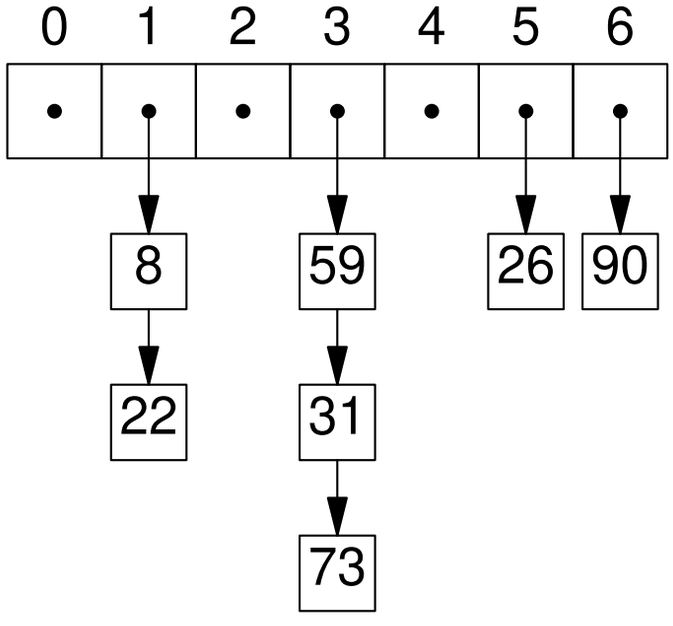
❒ Einfügen von $k = 59$ mit $h(k) = 59 \bmod 7 = 3$:



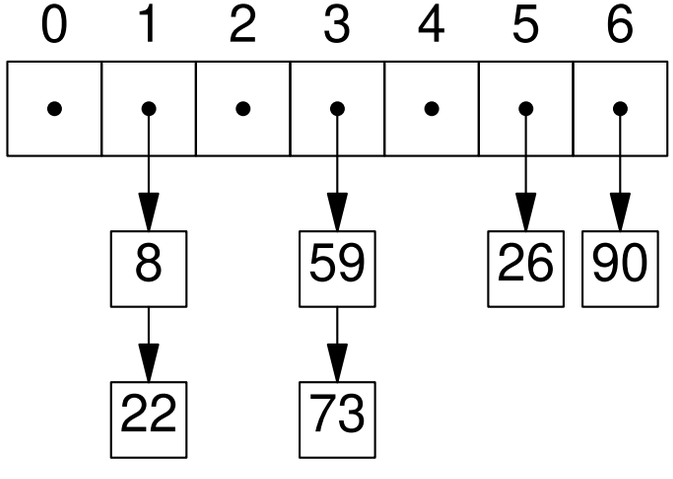
❒ Einfügen von $k = 90$ mit $h(k) = 90 \bmod 7 = 6$:



❒ Einfügen von $k = 8$ mit $h(k) = 8 \bmod 7 = 1$:



❒ Löschen von $k = 31$ mit $h(k) = 31 \bmod 7 = 3$:



2.5.3 Laufzeitanalyse

- Betrachte eine Tabelle der Größe N, die momentan m Objekte enthält, d. h. ihr *Belegungsfaktor* oder *Füllgrad* ist $\alpha = \frac{m}{N}$.

- α ist gleichzeitig die durchschnittliche Anzahl von Objekten pro Platz, d. h. die durchschnittliche Länge der gespeicherten Listen.

- Als Maß für die *Laufzeit* einer Operation (Einfügen/Ersetzen eines Objekts, Suchen und Löschen eines Schlüssels) wird die Anzahl der erforderlichen Schlüsselvergleiche verwendet.

Schlimmster Fall

- Alle Schlüssel besitzen den gleichen Streuwert/Index i.

  $\rightarrow$ $tab[i]$ enthält eine Liste mit m Objekten, die übrigen Plätze sind leer.

  $\rightarrow$ Bei jeder Operation muss die Liste $tab[i]$ u. U. vollständig durchlaufen werden

  $\rightarrow$ Laufzeit jeweils $O(m)$

Idealfall

- Eine Streuwertfunktion *streut ideal*, wenn jeder Index $0, \ldots, N-1$ mit der gleichen Wahrscheinlichkeit $\frac{1}{N}$ als Funktionswert auftritt.

  (Dies wird auch als Simple-uniform-hashing-Annahme bezeichnet.)

- Anmerkungen:
  - Trotzdem ist die Wahrscheinlichkeit, dass die Listen aller Plätze exakt gleich lang sind, äußerst gering.
    (Vergleich: Wenn man sehr oft mit einem idealen Würfel würfelt, ist die Wahrscheinlichkeit, dass jede Augenzahl exakt gleich oft auftritt, auch sehr gering.)
  - Und die Wahrscheinlichkeit, dass in einer Menge von Objekten mindestens zwei den gleichen Streuwert/Index besitzen, ist relativ hoch, auch wenn m deutlich kleiner als N ist.
    Vgl. Geburtstagsparadoxon: Befinden sich in einem Raum mindestens 23 Personen, dann ist die Wahrscheinlichkeit, dass zwei oder mehr dieser Personen am gleichen Tag (ohne Beachtung des Jahrganges) Geburtstag haben, größer als 50 %. (Wikipedia)

- Bei den folgenden Analysen wird eine ideal streuende Funktion vorausgesetzt.

Erfolglose Suche

Behauptung:

- Bei einer Suche nach einem Schlüssel k, der nicht in der Tabelle enthalten ist, werden im Durchschnitt α Schlüsselvergleiche ausgeführt.

Anmerkung:

- „Im Durchschnitt“ bedeutet:
  Wenn diese Operation für sehr viele Schlüssel k und sehr viele Tabellen mit unterschiedlicher Verteilung der Objekte ausgeführt wird, dann werden im Mittel α Schlüsselvergleiche pro Operation ausgeführt.

Beweis:

- Sei $i = h(k)$, wobei jeder Index i gleich wahrscheinlich ist.
- Die Länge der Liste $tab[i]$ ist im Durchschnitt α.
- Da die Liste kein Objekt mit Schlüssel k enthält, muss sie komplett durchsucht werden.
- Dabei werden im Durchschnitt α Schlüsselvergleiche ausgeführt.

Erfolgreiche Suche

Behauptung:

- Bei einer Suche nach einem Schlüssel k, der in der Tabelle enthalten ist, werden im Durchschnitt $1 + \frac{\alpha}{2} - \frac{1}{2N} \approx 1 + \frac{\alpha}{2}$ Schlüsselvergleiche ausgeführt.

Überprüfung der Behauptung z. B. für $m = 1$ und $m = 2$

Beweis:

- Sei $i = h(k)$.
- Um das Objekt x mit Schlüssel k zu finden, müssen alle Objekte überprüft werden, die sich in der Liste $tab[i]$ vor diesem Objekt befinden, sowie dieses Objekt selbst.
- Also hängt die Laufzeit von der Anzahl der Objekte ab, die später als x in die Tabelle bzw. in diese Liste eingefügt wurden (weil neue Objekte gemäß § 2.5.1 immer am Anfang einer Liste eingefügt werden).
- Für $j = 1, \ldots, m$ sei x_j das j-te Objekt, das in die Tabelle eingefügt wurde.

- Die Anzahl der Objekte, die später als x_j in die Tabelle eingefügt wurden, beträgt $m - j$.
- Die Wahrscheinlichkeit, dass eines dieser Objekte den gleichen Streuwert besitzt wie x_j, beträgt gemäß Simple-uniform-hashing-Annahme $\frac{1}{N}$.
- Daher ist die durchschnittliche Anzahl von Objekten, die später als x_j in die gleiche Liste eingefügt wurden, gleich $\frac{m-j}{N}$.
- Somit ist die durchschnittliche Laufzeit einer Suche nach dem Schlüssel von x_j gleich $1 + \frac{m-j}{N}$ $(j = 1, \ldots, m)$.
- Die durchschnittliche Laufzeit einer Suche nach dem Schlüssel irgendeines dieser Objekte x erhält man als Durchschnitt dieser Werte über alle j:

$$\frac{1}{m}\sum_{j=1}^{m}\left(1+\frac{m-j}{N}\right) = \frac{1}{m}\sum_{j=1}^{m}1 + \frac{1}{m}\sum_{j=1}^{m}\frac{m}{N} - \frac{1}{m}\sum_{j=1}^{m}\frac{j}{N} = 1 + \frac{m}{N} - \frac{1}{mN}\frac{m(m+1)}{2} =$$

$$1 + \frac{2m}{2N} - \frac{m+1}{2N} = 1 + \frac{m-1}{2N} = 1 + \frac{m}{2N} - \frac{1}{2N} = 1 + \frac{\alpha}{2} - \frac{1}{2N}$$

Einfügen und Ersetzen

- Beim Einfügen eines neuen Objekts (k, v) werden genauso viele Schlüsselvergleiche durchgeführt wie bei einer erfolglosen Suche nach dem Schlüssel k dieses Objekts.

- Beim Ersetzen eines bereits vorhandenen Objekts (k, v) werden genauso viele Schlüsselvergleiche durchgeführt wie bei einer erfolgreichen Suche nach dem Schlüssel k dieses Objekts.

Erfolgreiches und erfolgloses Löschen

- Beim Löschen eines vorhandenen Objekts (k, v) werden genauso viele Schlüsselvergleiche durchgeführt wie bei einer erfolgreichen Suche nach dem Schlüssel k dieses Objekts.

- Beim Löschen eines nicht vorhandenen Objekts (k, v) werden genauso viele Schlüsselvergleiche durchgeführt wie bei einer erfolglosen Suche nach dem Schlüssel k dieses Objekts.

2.6 Offene Adressierung (open addressing)

2.6.1 Grundoperationen

- Gegeben sei eine Reihe *tab* der Größe N zur Speicherung von Objekten sowie eine *Sondierungsfunktion* s, die jedem Schlüssel k eine Permutation $(s_0(k), \ldots, s_{N-1}(k))$ der Indexwerte $0, \ldots, N-1$ zuordnet.

- Hilfsoperation: Suchen des Platzes eines Schlüssels k
  1. Für $j = 0, \ldots, N-1$:
     1. Berechne den Index $i = s_j(k)$.
     2. Wenn $tab[i]$ leer ist, liefere „nicht vorhanden“ und entweder den gemerkten Index (falls es bereits einen gibt) oder (andernfalls) den Index i zurück.
     3. Wenn $tab[i]$ eine Löschmarkierung (siehe unten) enthält und bis jetzt noch kein Index gemerkt wurde, merke den Index i.
     4. Wenn $tab[i]$ ein Objekt (k', v') mit $k' = k$ und irgendeinem Wert v' enthält, liefere „vorhanden“ und den Index i zurück.
  2. Wenn während der Schleife ein Index gemerkt wurde, liefere „nicht vorhanden“ und diesen Index zurück.
  3. Andernfalls liefere „Tabelle voll“ zurück.

- Einfügen/Ersetzen eines Objekts (k, v) mit Schlüssel k und Wert v
  1. Führe die obige Hilfsoperation aus.
  2. Wenn sie einen Index i zurückliefert, speichere das Objekt (k, v) in $tab[i]$.
  3. Andernfalls signalisiere einen Fehler (Tabelle voll).

- Suchen eines Schlüssels k
  1. Führe die obige Hilfsoperation aus.
  2. Wenn sie „vorhanden“ und einen Index i zurückliefert,
     liefere den Wert v' des in $tab[i]$ gespeicherten Objekts (k', v') zurück.
  3. Andernfalls liefere $\perp$.

- Löschen eines Schlüssels k
  1. Führe die obige Hilfsoperation aus.
  2. Wenn sie „vorhanden“ und einen Index i zurückliefert,
     schreibe eine Löschmarkierung in $tab[i]$.

2.6.2 Anmerkungen

- Anders als bei Verkettung, wo eine Tabelle prinzipiell beliebig viele Objekte aufnehmen kann, ist die Kapazität bei offener Adressierung durch die Tabellengröße N beschränkt.

- Daraus folgt, dass der Belegungsfaktor bei offener Adressierung nicht größer als 1 sein kann.

- Je voller die Tabelle ist, desto mehr Sondierungsschritte sind im Durchschnitt nötig, um noch einen freien Platz zur Speicherung eines Objekts zu finden.

- Löschmarkierungen werden gebraucht, damit ein damit markierter Platz beim Einfügen wie ein leerer Platz behandelt wird (der wieder belegt werden kann), beim Suchen aber wie ein belegter Platz (bei dem noch weitergesucht werden muss).

2.6.3 Lineare Sondierung (linear probing)

- Gegeben sei eine Tabelle der Größe N sowie eine Streuwertfunktion h.

- Definiere $s_j(k) = (h(k) + j) \bmod N$ für $j = 0, \ldots, N-1$,
  d. h. die Sondierungssequenz $s(k)$ eines Schlüssels k beginnt mit seinem Streuwert $h(k)$ (eingeschränkt auf den Wertebereich $\{0, \ldots, N-1\}$)
  und durchläuft dann der Reihe nach alle weiteren Plätze der Tabelle.

- Problem der direkten Verstopfung (primary clustering problem):
  - Wenn zwei oder mehr Objekte den gleichen Streuwert i besitzen (was auch bei idealer Verteilung der Streuwerte relativ häufig auftritt, vgl. § 2.5.3), entsteht an dieser Stelle der Tabelle ein „Klumpen“ (cluster) von Objekten.
  - Jedes weitere Objekt, dessen Streuwert in diesem Klumpen liegt, vergrößert diesen, selbst wenn sein Streuwert verschieden von i ist.
  - Je größer ein derartiger Klumpen ist, desto höher ist die Laufzeit aller Operationen für die betroffenen Objekte.

- Beispiel: $N = 16$, $h(k) = k$
  Einfügen von 0, 16, 32, 48, 64, 80, 1, 2, 3, 4, 5, 6

2.6.4 Quadratische Sondierung (quadratic probing)

- Gegeben sei eine Tabelle der Größe N sowie eine Streuwertfunktion h.

- Definiere $s_j(k) = \left(h(k) + \frac{j + j^2}{2} \right) \bmod N$ für $j = 0, \ldots, N-1$,

  d. h. die Sondierungssequenz $s(k)$ eines Schlüssels k beginnt wieder mit seinem Streuwert $h(k)$ (eingeschränkt auf den Wertebereich $\{ 0, \ldots, N-1 \}$) und durchläuft dann der Reihe nach die Plätze mit Abstand 1, 3, 6, 10, ... von diesem Anfangsplatz.

- Praktische Berechnung:

$$s_j(k) = \left(h(k) + \frac{j + j^2}{2} \right) \bmod N = \left(h(k) + \frac{j\,(j+1)}{2} \right) \bmod N = \left(h(k) + \sum_{i=1}^{j} i \right) \bmod N =$$

$$= \begin{cases} h(k) \bmod N & \text{für } j = 0 \\ \left(h(k) + \sum_{i=1}^{j-1} i + j \right) \bmod N = \left(s_{j-1}(k) + j \right) \bmod N & \text{für } j = 1, \ldots, N-1 \end{cases}$$

- Zu zeigen:
  Die Sondierungssequenz $(s_0(k), \ldots, s_{N-1}(k))$ ist für geeignete Werte von N tatsächlich eine Permutation der Indexwerte $0, \ldots, N-1$, d. h. jeder Indexwert tritt in der Sequenz genau einmal auf.

- Beweis für Zweierpotenzen $N = 2^p$ mit $p \in \mathbb{N}_0$:
  - Zu zeigen: Für alle $i, j \in \{0, \ldots, N-1\}$ mit $i \neq j$ gilt: $s_i(k) \neq s_j(k)$.
  - Annahme: Es gibt $i, j \in \{0, \ldots, N-1\}$ mit $i \neq j$ (o. B. d. A. $i < j$), für die gilt: $s_i(k) = s_j(k)$.
  - Das heißt:

$$h(k) + \frac{i + i^2}{2} \equiv h(k) + \frac{j + j^2}{2} \pmod{N}$$

$$\Leftrightarrow \frac{i + i^2}{2} \equiv \frac{j + j^2}{2} \pmod{N}$$

$$\Leftrightarrow \frac{j + j^2}{2} - \frac{i + i^2}{2} = c\,N = c\,2^p \text{ für ein } c \in \mathbb{N}$$

$$\Leftrightarrow c\,2^{p+1} = j + j^2 - i - i^2 = (j - i)(i + j + 1)$$

$$\Leftrightarrow (j - i)(i + j + 1) \text{ ist durch } 2^{p+1} \text{ teilbar}$$

- Fallunterscheidung:
  - Wenn i und j entweder beide gerade oder beide ungerade sind, ist der Faktor $i + j + 1$ ungerade und somit nicht durch 2 teilbar. Also muss der andere Faktor $j - i$ durch 2^{p+1} teilbar sein. Wegen $i < j$ sowie $j \leq N - 1$ und $i \geq 0$ gilt jedoch: $0 < j - i \leq (N - 1) - 0 = N - 1 = 2^p - 1 < 2^{p+1}$, also kann $j - i$ nicht durch 2^{p+1} teilbar sein.
  - Wenn i gerade und j ungerade ist oder umgekehrt, ist der Faktor $j - i$ ungerade und somit nicht durch 2 teilbar. Also muss der andere Faktor $i + j + 1$ durch 2^{p+1} teilbar sein. Wegen $0 \leq i \leq N - 1$ und $0 \leq j \leq N - 1$ gilt jedoch: $0 < i + j + 1 \leq (N - 1) + (N - 1) + 1 = 2N - 1 = 2^{p+1} - 1 < 2^{p+1}$, also kann $i + j + 1$ nicht durch 2^{p+1} teilbar sein.
- Da alle möglichen Fälle zum Widerspruch führen, muss die obige Annahme falsch und damit die Behauptung richtig sein.

- Quadratische Sondierung vermeidet das Problem der direkten Verstopfung:
  - Wenn zwei oder mehr Objekte den gleichen Streuwert i besitzen, werden sie in den Plätzen i, $i+1$, $i+3$, $i+6$, ... (mod N) gespeichert, d. h. die Plätze $i+2$, $i+4$, $i+5$, ... (mod N) bleiben frei, womit die „Verstopfung“ der gesamten Nachbarschaft des Platzes i vermieden wird.
  - Somit können Objekte mit diesen Streuwerten normalerweise direkt in diesen Plätzen gespeichert werden (sofern sie nicht bereits durch andere Objekte mit gleichem Streuwert belegt sind).
  - Objekte mit Streuwerten $i+1$, $i+3$, $i+6$, ... (mod N) können normalerweise in den direkten Nachbarplätzen $i+2$, $i+4$, $i+7$, ... (mod N) gespeichert werden.
- Problem der indirekten Verstopfung (secondary clustering problem):
  - Trotzdem führen viele Objekte mit gleichem Streuwert i nach wie vor zur Bildung einer „Kette“ i, $i+1$, $i+3$, $i+6$, ... (mod N), die bei Operationen für diese Objekte jedesmal ganz oder teilweise durchlaufen werden muss.
- Beispiel: $N = 16$, $h(k) = k$
  Einfügen von 0, 16, 32, 48, 64, 80, 1, 2, 3, 4, 5, 6
  (vgl. § 2.6.3)

2.6.5 Doppelte Streuung (double hashing)

- Gegeben sei eine Tabelle der Größe N sowie zwei Streuwertfunktionen h_1 und h_2.

- Definiere $s_j(k) = (h_1(k) + j\,h_2(k)) \bmod N$ für $j = 0, \ldots, N-1$,
  d. h. die Sondierungssequenz $s(k)$ eines Schlüssels k beginnt mit dem Streuwert $h_1(k)$ (eingeschränkt auf den Wertebereich $\{0, \ldots, N-1\}$) und durchläuft dann der Reihe nach die Plätze mit Abstand $h_2(k)$, $2\,h_2(k)$, $3\,h_2(k)$, ... von diesem Anfangsplatz.

- Damit ist doppelte Streuung ähnlich zu linearer Sondierung, aber mit einer „Schrittweite“, die vom Schlüssel k abhängt.

- Damit werden normalerweise beide Verstopfungs-Probleme vermieden:
  - Wenn zwei oder mehr Objekte den gleichen h_1-Streuwert i besitzen, besitzen sie in der Regel unterschiedliche h_2-Streuwerte und somit auch unterschiedliche Sondierungssequenzen.
  - Somit entsteht an der Stelle i normalerweise weder ein Klumpen noch eine Kette.

- Beispiel: $N = 16$, $h_1(k) = k$, $h_2(k) = k \bmod 15 + 1 - k \bmod 15 \bmod 2$
  Einfügen von 0, 16, 32, 48, 64, 80, 1, 2, 3, 4, 5, 6
  (vgl. § 2.6.3 und § 2.6.4)

- Zu zeigen:
  Die Sondierungssequenz $(s_0(k), \ldots, s_{N-1}(k))$ ist unter geeigneten Bedingungen tatsächlich eine Permutation der Indexwerte $0, \ldots, N-1$, d. h. jeder Indexwert tritt in der Sequenz genau einmal auf.

- Beweis für den Fall, dass $h_2(k)$ und N für alle Schlüssel k teilerfremd sind:
  - Zu zeigen: Für alle $i, j \in \{0, \ldots, N-1\}$ mit $i \neq j$ gilt: $s_i(k) \neq s_j(k)$.
  - Annahme: Es gibt $i, j \in \{0, \ldots, N-1\}$ mit $i \neq j$, für die gilt: $s_i(k) = s_j(k)$.
  - Das heißt:
    $h_1(k) + i\, h_2(k) \equiv h_1(k) + j\, h_2(k) \pmod{N}$
    $\Leftrightarrow i\, h_2(k) \equiv j\, h_2(k) \pmod{N}$
    $\Leftrightarrow i \equiv j \pmod{N}$ (da $h_2(k)$ und N teilerfremd sind und $h_2(k)$ deshalb ein multiplikatives Inverses modulo N besitzt)
    $\Leftrightarrow i = j$ (da $i, j \in \{0, \ldots, N-1\}$)
  - Widerspruch!

- Anmerkung:
  Damit $h_2(k)$ und N teilerfremd sind, darf $h_2(k)$ insbesondere nicht 0 sein.

- Praktische Realisierungsmöglichkeit 1:
  - Die Tabellengröße N ist eine Zweierpotenz 2^p mit $p \in \mathbb{N}_0$.
  - Die Streuwertfunktion h_2 liefert nur ungerade Zahlen.
  - Dann sind $h_2(k)$ und N für alle Schlüssel k teilerfremd.

- Beispiel:
  - $N = 2^{10} = 1024$
  - $h_1(k) = k$
  - $h_2(k) = 2k + 1$

- Für $k = 123456$ ergibt sich zum Beispiel:
  - $h_1(k) = 123456 \equiv 576 \pmod{1024}$
  - $h_2(k) = 2 \cdot 123456 + 1 = 246913 \equiv 129 \pmod{1024}$
  - $s_j(k) = (123456 + j \cdot 246913) \bmod 1024 = (576 + j \cdot 129) \bmod 1024$ für $j = 0, \ldots, 1023$

- Praktische Realisierungsmöglichkeit 2:
  - Die Tabellengröße N ist eine Primzahl.
  - Die Zahl N' ist „etwas kleiner“ als N, z. B. $N' = N - 1$.
  - $h_1(k) = k$
  - $h_2(k) = k \bmod N' + 1$
  - Wegen $k \bmod N' \in \{0, \ldots, N' - 1\}$ und $N' \leq N - 1$ ist $h_2(k) \in \{1, \ldots, N'\} \subseteq \{1, \ldots, N - 1\}$.
  - Da N eine Primzahl ist, sind somit alle Werte von $h_2(k)$ teilerfremd zu N.
- Beispiel:
  - $N = 701$, $N' = 700$
- Für $k = 123456$ ergibt sich zum Beispiel:
  - $h_1(k) = 123456 \equiv 80 \pmod{701}$
  - $h_2(k) = 123456 \bmod 700 + 1 = 257$
  - $s_j(k) = (123456 + j \cdot 257) \bmod 701 = (80 + j \cdot 257) \bmod 701$ für $j = 0, \ldots, 700$

2.6.6 Laufzeitanalyse

Idealfall

- Eine Sondierungsfunktion *streut ideal*, wenn jede Permutation der Indexwerte $\{0, \ldots, N-1\}$ mit der gleichen Wahrscheinlichkeit $\frac{1}{N!}$ als Funktionswert auftritt.
  (Dies wird auch als Uniform-hashing-Annahme bezeichnet.)

- Bei linearer und quadratischer Sondierung liefert die Sondierungsfunktion maximal N verschiedene Permutationen, die jeweils mit Wahrscheinlichkeit $\frac{1}{N}$ auftreten, sofern die zugrundeliegende Streuwertfunktion ideal streut.

- Bei doppelter Streuung liefert die Sondierungsfunktion maximal $N(N-1)$ verschiedene Permutationen, d. h. auch hier ist man theoretisch weit von einer idealen Streuung entfernt.
  Trotzdem funktioniert doppelte Streuung erfahrungsgemäß sehr gut.

- Bei den folgenden Analysen wird eine ideal streuende Sondierungsfunktion vorausgesetzt, obwohl diese Annahme in der Praxis nicht wirklich zutreffend ist.

- Außerdem wird wieder eine Tabelle der Größe N betrachtet, die momentan m Objekte enthält, d. h. $\alpha = \frac{m}{N}$.

Erfolglose Suche

Behauptung

- Bei einer Suche nach einem Schlüssel k, der nicht in der Tabelle enthalten ist, werden im Durchschnitt höchstens $\frac{\alpha}{1-\alpha}$ Schlüsselvergleiche ausgeführt.

Anmerkungen

- Die Behauptung ist nur für $\alpha < 1$ sinnvoll.
- Die Formulierung „höchstens“ bedeutet nicht, dass sich die Aussage auf den schlimmsten Fall bezieht, sondern dass die *durchschnittliche* Anzahl der Schlüsselvergleiche auf jeden Fall nicht größer als $\frac{\alpha}{1-\alpha}$ ist.

Beweis

- Bei der Suche nach einem Objekt mit Schlüssel k werden so lange Schlüssel aus der Tabelle mit k verglichen, bis man auf einen leeren Platz trifft.
  (Wegen $\alpha < 1$ gibt es mindestens einen leeren Platz.)
- Beim ersten Sondierungsversuch sind von insgesamt N Plätzen m Plätze belegt.

- Beim zweiten Sondierungsversuch (der nur ausgeführt wird, wenn der Platz beim ersten Versuch belegt ist) sind von insgesamt $N-1$ verbleibenden Plätzen $m-1$ belegt.

- Beim dritten Sondierungsversuch (der nur ausgeführt wird, wenn die Plätze beim ersten und zweiten Versuch belegt sind) sind von insgesamt $N-2$ verbleibenden Plätzen $m-2$ belegt.

- Usw.

- Beim j-ten Sondierungsversuch (der nur ausgeführt wird, wenn die Plätze bei allen vorhergehenden Versuchen belegt sind) sind von insgesamt $N-j+1$ verbleibenden Plätzen $m-j+1$ belegt.

- Daraus folgt: Die Wahrscheinlichkeit $\mathrm{P}(V \geq j)$, dass die Anzahl V der Schlüsselvergleiche mindestens j ist, d. h. dass die Plätze der ersten j Sondierungsversuche belegt sind, beträgt für $j = 1, \ldots, m$:

  $$\mathrm{P}(V \geq j) = \frac{m}{N} \cdot \frac{m-1}{N-1} \cdot \frac{m-2}{N-2} \cdots \frac{m-j+1}{N-j+1} \leq \left(\frac{m}{N}\right)^j = \alpha^j$$

  (Beachte: Für $0 \leq i \leq m < N$ gilt $\frac{m-i}{N-i} \leq \frac{m}{N}$.
  Aufgrund der Uniform-hashing-Annahme hängt der Ausgang eines Sondierungsversuchs nicht von den vorhergehenden Versuchen ab.)

- Für $j = 0$ gilt offensichtlich („sicheres Ereignis“): $\mathrm{P}(V \geq j) = \mathrm{P}(V \geq 0) = 1 = \alpha^0 = \alpha^j$
- Für $j > m$ gilt offensichtlich („unmögliches Ereignis“): $\mathrm{P}(V \geq j) = 0 \leq \alpha^j$
- Somit gilt für alle $j = 0, 1, 2, \ldots$: $\mathrm{P}(V \geq j) \leq \alpha^j$
- Für die Wahrscheinlichkeit $\mathrm{P}(V = j)$, dass die Anzahl V der Schlüsselvergleiche gleich j ist, gilt offensichtlich für $j = 0, 1, 2, \ldots$: $\mathrm{P}(V = j) = \mathrm{P}(V \geq j) - \mathrm{P}(V \geq j + 1)$
- Für die durchschnittliche Anzahl $\mathrm{E}(V)$ der Schlüsselvergleiche („Erwartungswert von V“) gilt somit:

$$\mathrm{E}(V) = \sum_{j=0}^{\infty} j \cdot \mathrm{P}(V = j) = \sum_{j=0}^{\infty} j \cdot (\mathrm{P}(V \geq j) - \mathrm{P}(V \geq j + 1)) =$$

$$\sum_{j=0}^{\infty} j \cdot \mathrm{P}(V \geq j) - \sum_{j=0}^{\infty} j \cdot \mathrm{P}(V \geq j + 1) = \sum_{j=0}^{\infty} j \cdot \mathrm{P}(V \geq j) - \sum_{j=1}^{\infty} (j - 1) \cdot \mathrm{P}(V \geq j) =$$

$$0 \cdot \mathrm{P}(V \geq 0) + \sum_{j=1}^{\infty} (j - (j - 1)) \cdot \mathrm{P}(V \geq j) = \sum_{j=1}^{\infty} \mathrm{P}(V \geq j) \leq \sum_{j=1}^{\infty} \alpha^j = \frac{\alpha}{1 - \alpha}$$

Einfügen

- Beim Einfügen eines neuen Objekts (k, v) wird die gleiche Folge von Sondierungsversuchen durchlaufen und damit auch die gleiche Anzahl von Schlüsselvergleichen ausgeführt wie bei einer erfolglosen Suche nach dem Schlüssel k.

- Deshalb beträgt die durchschnittliche Anzahl von Schlüsselvergleichen bei einer solchen Operation ebenfalls höchstens $\frac{\alpha}{1-\alpha}$.

Erfolgreiche Suche

Behauptung

- Bei einer Suche nach einem Schlüssel k, der in der Tabelle enthalten ist, werden im Durchschnitt höchstens $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ Schlüsselvergleiche ausgeführt.

Beweis

- Bei der Suche nach dem Objekt mit Schlüssel k wird wiederum die gleiche Folge von Sondierungsversuchen durchlaufen wie beim Einfügen dieses Objekts.
  Allerdings findet man beim letzten Sondierungsversuch keinen leeren Platz, sondern das gesuchte Objekt, sodass ein Schlüsselvergleich mehr ausgeführt wird.

- Wenn dieses Objekt das $(j+1)$-te eingefügte Objekt ist ($j = 0, \ldots, m-1$), war der Belegungsfaktor α zum Zeitpunkt seiner Einfügung gleich $\frac{j}{N}$.

- Somit wurden bei dieser Einfügung im Durchschnitt höchstens $\frac{\frac{j}{N}}{1-\frac{j}{N}} = \frac{j}{N-j}$ Schlüsselvergleiche ausgeführt.

- Also werden bei einer Suche nach diesem Objekt im Durchschnitt höchstens $\frac{j}{N-j} + 1 = \frac{N}{N-j}$ Schlüsselvergleiche ausgeführt.

- Der Durchschnittswert über alle bis jetzt eingefügten Objekte beträgt:

$$\frac{1}{m}\sum_{j=0}^{m-1}\frac{N}{N-j} = \frac{N}{m}\sum_{j=0}^{m-1}\frac{1}{N-j} = \frac{1}{\alpha}\sum_{i=N-m+1}^{N}\frac{1}{i} \leq \frac{1}{\alpha}\int_{N-m}^{N}\frac{1}{x}\,\mathrm{d}x = \frac{1}{\alpha}\Big[\ln x\Big]_{N-m}^{N} =$$

$$\frac{1}{\alpha}\left(\ln N - \ln(N-m)\right) = \frac{1}{\alpha}\ln\frac{N}{N-m} = \frac{1}{\alpha}\ln\frac{1}{1-\frac{m}{N}} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

(Für die Abschätzung wird das aus der Analysis bekannte Integralkriterium verwendet.)

Ersetzen

- Beim Ersetzen eines bereits vorhandenen Objekts (k, v) werden genauso viele Schlüsselvergleiche durchgeführt wie bei einer erfolgreichen Suche nach dem Schlüssel k dieses Objekts, d. h. höchstens $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$.

Erfolgreiches und erfolgloses Löschen

- Beim Löschen eines vorhandenen Objekts (k, v) werden genauso viele Schlüsselvergleiche durchgeführt wie bei einer erfolgreichen Suche nach dem Schlüssel k dieses Objekts.

- Beim Löschen eines nicht vorhandenen Objekts (k, v) werden genauso viele Schlüsselvergleiche durchgeführt wie bei einer erfolglosen Suche nach dem Schlüssel k dieses Objekts.

2.7 Laufzeitvergleich

- Die folgende Tabelle zeigt die durchschnittliche Anzahl von Schlüsselvergleichen bei einer erfolglosen bzw. erfolgreichen Suche in Abhängigkeit vom Belegungsfaktor α bei Verwendung von Verkettung bzw. offener Adressierung, jeweils unter der entsprechenden Uniform-hashing-Annahme.

- Bei den Formeln für offene Adressierung handelt es sich jedoch um Abschätzungen nach oben, d. h. die tatsächlichen Zahlenwerte sind u. U. deutlich kleiner.

α	Verkettung		Offene Adressierung	
	erfolglos	erfolgreich	erfolglos	erfolgreich
	α	$1 + \frac{\alpha}{2}$	$\frac{\alpha}{1-\alpha}$	$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
0.1	0.1	1.05	0.11111	1.05361
0.2	0.2	1.1	0.25	1.11572
0.5	0.5	1.25	1	1.38629
0.75	0.75	1.375	3	1.84839
0.9	0.9	1.45	9	2.55843
0.95	0.95	1.475	19	3.15340

2.8 Vergrößern und Verkleinern von Streuwerttabellen

2.8.1 Arithmetische Vergrößerung von Reihen

- Gegeben sei eine Reihe mit Anfangsgröße $N_1 = N$, die sukzessive gefüllt wird.
- Wenn sie voll ist, wird sie um N Elemente auf $N_2 = 2\,N$ „vergrößert“, indem eine neue Reihe der Größe N_2 beschafft und die bereits vorhandenen N_1 Elemente umkopiert werden.
- Wenn diese Reihe wieder voll ist, wird sie erneut um N Elemente auf $N_3 = 3\,N$ vergrößert, wobei die jetzt vorhandenen N_2 Elemente umkopiert werden müssen.
- Usw.
- Unmittelbar vor der k-ten Vergrößerung besitzt die Reihe die Größe $N_k = k\,N$, und es mussten insgesamt $N_1 + \ldots + N_{k-1} = \sum_{j=1}^{k-1} N_j = \sum_{j=1}^{k-1} j\,N = \frac{k\,(k-1)}{2}\,N$ Elemente kopiert werden.

- Verteilt man diesen Gesamtkopieraufwand gleichmäßig auf die $k\,N$ Elemente, ergibt sich als *amortisierter Aufwand* für das Hinzufügen eines einzelnen Elements

$$\frac{\frac{k\,(k-1)}{2}\,N}{k\,N} = \frac{k-1}{2}\,,$$

d. h. dieser Aufwand wird umso größer, je öfter die Reihe vergrößert wird.

2.8.2 Geometrische Vergrößerung von Reihen

- Gegeben sei eine Reihe mit Anfangsgröße $N_0 = N$, die sukzessive gefüllt wird.
- Wenn sie voll ist, wird sie um den Faktor q auf $N_1 = q\,N$ vergrößert, wobei die bereits vorhandenen N_0 Elemente umkopiert werden müssen.
- Wenn sie wieder voll ist, wird sie erneut um den Faktor q auf $N_2 = q^2\,N$ vergrößert, wobei die jetzt vorhandenen N_1 Elemente umkopiert werden müssen.
- Usw.
- Unmittelbar vor der $(k+1)$-ten Vergrößerung besitzt die Reihe die Größe $N_k = q^k\,N$, und es mussten insgesamt

  $$N_0 + \ldots + N_{k-1} = \sum_{j=0}^{k-1} N_j = \sum_{j=0}^{k-1} q^j\,N = N \sum_{j=0}^{k-1} q^j = N\,\frac{q^k - 1}{q-1} < N\,\frac{q^k}{q-1}$$

  Elemente kopiert werden.

- Verteilt man diesen Gesamtkopieraufwand wieder gleichmäßig auf die $q^k N$ Elemente, ergibt sich als amortisierter Aufwand für das Hinzufügen eines einzelnen Elements weniger als $\frac{N \frac{q^k}{q-1}}{q^k N} = \frac{1}{q-1}$,

  d. h. dieser Aufwand bleibt unabhängig von der Anzahl der Vergrößerungen konstant. (Der Kopieraufwand für eine einzelne Vergrößerung wird natürlich trotzdem jedesmal größer.)

2.8.3 Einfaches Vergrößern und Verkleinern von Streuwerttabellen

- Wenn der Belegungsfaktor einer Streuwerttabelle einen bestimmten kritischen Wert (typischerweise etwa 3/4, vgl. die Zahlenwerte in § 2.7) überschreitet, ist es ratsam, die Tabelle zu vergrößern.
- Aufgrund der vorangegangenen Überlegungen in § 2.8.1 und § 2.8.2 ist eine (in etwa) geometrische Vergrößerung sinnvoll (ggf. unter Beachtung der Kriterien für „gute“ Tabellengrößen in § 2.3).
- Dabei muss auch die Streuwertfunktion an die neue Tabellengröße angepasst werden.
- Beim Umkopieren der vorhandenen Objekte muss ihr Platz in der neuen Tabelle jeweils mit der neuen Streuwertfunktion berechnet werden, was den Aufwand für das Vergrößern nochmals erhöht.
- Wenn der Belegungsfaktor kleiner als ein bestimmter Wert wird, kann entsprechend auch eine Verkleinerung der Tabelle sinnvoll sein, um Speicherplatz zu sparen.
- Dabei ist jedoch darauf zu achten, dass die verkleinerte Tabelle wieder ausreichend Platz für neue Objekte besitzt, bevor sie erneut vergrößert werden muss, um ein ständiges Pendeln zwischen Vergrößern und Verkleinern auch unter ungünstigen Bedingungen zu vermeiden.

2.8.4 Inkrementelles Vergrößern und Verkleinern von Streuwerttabellen

- Obwohl der amortisierte Aufwand jeder Einfügeoperation bei geometrischer Vergrößerung konstant ist, ist der tatsächliche Aufwand derjenigen Einfügeoperation, die zu einer Vergrößerung der Tabelle führt, proportional zur momentanen Tabellengröße und damit potentiell sehr hoch.

- Dies kann insbesondere für interaktive und Echtzeitanwendungen problematisch sein.

- Beim inkrementellen Vergrößern wird die alte (kleine) Tabelle zunächst aufbewahrt und das aufwendige Umkopieren der Objekte schrittweise ausgeführt:
  - Neue Objekte werden mit der neuen Streuwertfunktion in die neue (vergrößerte) Tabelle eingefügt.
  - Bei jeder solchen Einfügung werden außerdem ein paar weitere Objekte von der alten in die neue Tabelle umkopiert.
  - Wenn alle Objekte umkopiert wurden, kann die alte Tabelle freigegeben werden.
  - Um ein Objekt zu suchen oder zu löschen, muss es ggf. in der alten und in der neuen Tabelle (jeweils mit der zugehörigen Streuwertfunktion) gesucht werden.

- Um sicherzustellen, dass alle Objekte umkopiert wurden, bevor die (neue) Tabelle erneut vergrößert werden muss, muss die Anzahl der umkopierten Objekte pro Einfügeoperation geeignet gewählt werden.

- Zum Beispiel: Wenn die Tabellengröße bei jeder Vergrößerung verdoppelt wird, muss bei jeder Einfügeoperation mindestens ein weiteres Objekt umkopiert werden.

2.9 Ausblick

- (Minimale) perfekte Streuwertfunktionen
- Kryptographische Streuwertfunktionen