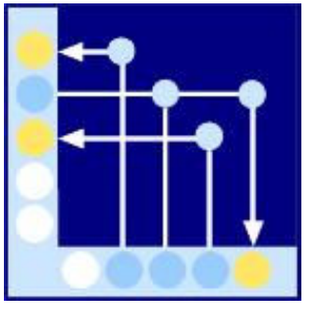
Fakultät Elektronik und Informatik
Studiengang Informatik



Algorithmen und Datenstrukturen 2

Vorlesung im Sommersemester 2018

Prof. Dr. habil. Christian Heinlein

christian.heinleins.net

1 Einleitung und Überblick

1.1 Vorlesungsüberblick

- Streuwerttabellen (hash tables)
  - Implementierung mit Verkettung (chaining)
  - Implementierung mit offener Adressierung (open addressing)
- Vorrangwarteschlangen (priority queues)
  - Implementierung mit binären Halden (binary heaps)
  - Implementierung mit Binomial-Halden (binomial heaps)
- Programmiertechniken
  - Nächstbest-Algorithmen (greedy algorithms)
  - Tabellengestützte Programmierung (dynamic programming)

- Elementare Graphalgorithmen
  - Breitensuche (breadth-first search)
  - Tiefensuche (depth-first search)
  - Starke Zusammenhangskomponenten (strongly connected components)
- Minimale Spannbäume (minimum spanning trees)
  - Algorithmus von Kruskal
  - Algorithmus von Prim
- Kürzeste Wege in Graphen (shortest path algorithms)
  - Algorithmus von Bellman und Ford
  - Algorithmus von Dijkstra
  - Algorithmus von Floyd und Warshall
- Anwendung mathematischer Methoden
  - Korrektheitsbeweise
  - Laufzeitabschätzungen

1.2 Organisatorisches

- Vorlesung
  - Folien sind online verfügbar
    - in der Regel kapitelweise, möglichst vor Beginn des jeweiligen Kapitels
    - bei Bedarf nachträgliche Ergänzungen oder Korrekturen
  - Beweise werden zum Teil an der Tafel entwickelt
  - Programme werden zum Teil interaktiv am Rechner entwickelt
- „Papier und Bleistift“-Übungsaufgaben zu jedem Kapitel
- 3 größere Programmieraufgaben während des Semesters
  als Teil der Prüfungsleistung (Anteil 1/3)
- 90-minütige Klausur im Prüfungszeitraum (Anteil 2/3)
- Voraussetzungen
  - Keine formalen Voraussetzungen
  - Aber die Inhalte folgender Vorlesungen werden inhaltlich vorausgesetzt:
    - Mathematik-Vorlesungen des 1. und 2. Semesters
    - Strukturierte und Objektorientierte Programmierung
    - Algorithmen und Datenstrukturen 1

1.3 Literaturhinweise

- T. H. Cormen, C. E. Leiserson, R. Rivest, C. Stein: *Algorithmen – Eine Einführung* (3. Auflage). R. Oldenbourg Verlag, München, 2010.

  Stellenweise werden Sachverhalte in dieser Vorlesung jedoch bewusst anders dargestellt als in diesem Standardwerk.

- R. Sedgewick, K. Wayne: *Algorithmen* (4., aktualisierte Auflage). Pearson, Hallbergmoos, 2010.

- U. Schöning: *Algorithmik*. Spektrum Akademischer Verlag, Heidelberg, 2001.

- D. E. Knuth: *The Art of Computer Programming* (Band 1 bis 3). Addison-Wesley, Reading, MA, 2001.

- https://de.wikipedia.org
  https://en.wikipedia.org

  Wikipedia ist häufig eine nützliche Informationsquelle.
  Stellenweise sind die Artikel jedoch ungenau oder widersprüchlich, z. B. beim Thema Graphen.

2 Streuwerttabellen (hash tables)

2.1 Einleitung und Motivation

2.1.1 Aufgabe

- Schreiben Sie ein Programm (z. B. in 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

```
import java.io.*;

// Listenelement.
class Elem {
  String word;            // Wort.
  int count;              // Häufigkeit.
  Elem next;              // Verkettung.

  Elem (String w, Elem n) {
    word = w; count = 1; next = n;
  }
}

// Hauptprogramm.
class WordCount {
  public static void main (String [] args) throws IOException {
    BufferedReader r =  // Standardeingabe als BufferedReader.
            new BufferedReader(new InputStreamReader(System.in));
    Elem h = null;      // Listenanfang (head).
    String w;           // Aktuelles Wort.
```

```
    // Eingabe zeilenweise lesen und verarbeiten.
    input:   // readLine kann IOException werfen.
    while ((w = r.readLine()) != null) {
      // Element e mit Wort w suchen.
      for (Elem e = h; e != null; e = e.next) {
        // Wenn vorhanden, Zähler erhöhen.
        if (e.word.equals(w)) {
            e.count++;
            continue input;
        }
      }
      // Andernfalls neues Element mit Zähler 1 erzeugen.
      h = new Elem(w, h);
    }

    // Ausgabe produzieren.
    for (Elem e = h; e != null; e = e.next) {
      System.out.println(e.count + " " + e.word);
    }
  }
}
```

- Problem: Linearer Aufwand für jede Suche

2.1.3 Lösung mit Suchbaum

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

2.1.4 Lösung mit Streuwerttabelle

```
import java.io.*;

// Tabellenelement.
class Elem {
  String word;           // Wort.
  int count;             // Häufigkeit.

  Elem (String w) {
    word = w; count = 1;
  }
}

// Hauptprogramm.
class WordCount {
  public static void main (String [] args) throws IOException {
    BufferedReader r =  // Standardeingabe als BufferedReader.
            new BufferedReader(new InputStreamReader(System.in));
    Elem [] tab = new Elem [1000]; // Tabelle.
    String w;           // Aktuelles Wort.
```

```
      // Eingabe zeilenweise lesen und verarbeiten.
      // readLine kann IOException werfen.
      while ((w = r.readLine()) != null) {
          // Streuwert des Worts als Index in die Tabelle verwenden.
          int i = w.hashCode() % tab.length;
          if (i < 0) i = -i;
          Elem e = tab[i];
          if (e != null) {
            // Wenn dort bereits ein Element ist, Zähler erhöhen.
            e.count++;
          }
          else {
            // Andernfalls neues Element mit Zähler 1 erzeugen.
            tab[i] = new Elem(w);
          }
      }

      // Ausgabe produzieren.
      for (Elem e : tab) if (e != null) {
        System.out.println(e.count + " " + e.word);
      }
    }
}
```

Diskussion

- Vorteile
  - Konstanter Aufwand für jede „Suche“ $\rightarrow$ maximal effizient
  - Kompakte Speicherung der Tabelle als Feld (array)
- 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`).
- Fragen
  - Wie löst man dieses Problem?
  - Wie berechnet man `hashCode` sinnvoll?

2.1.5 Lösung mit `java.util.HashMap`

```
import java.io.*;
import java.util.*;

// Hauptprogramm.
class WordCount {
  public static void main (String [] args) throws IOException {
    BufferedReader r =  // Standardeingabe als BufferedReader.
            new BufferedReader(new InputStreamReader(System.in));
    Map<String, Integer> tab = // Tabelle.
            new HashMap<String, Integer>(1000);
    String w;           // Aktuelles Wort.

    // Eingabe zeilenweise lesen und verarbeiten.
    // readLine kann IOException werfen.
    while ((w = r.readLine()) != null) {
        // Entweder einen neuen Eintrag mit Zähler 1 einfügen
        // oder den Zähler des vorhandenen Eintrags um 1 erhöhen.
        Integer c = tab.get(w);
        if (c == null) c = 0;
        tab.put(w, c + 1);
    }
```

```
    // Ausgabe produzieren.
    for (Map.Entry<String, Integer> e : tab.entrySet()) {
      System.out.println(e.getValue() + " " + e.getKey());
    }
  }
}
```

2.1.6 Lösung in C++11 mit `std::unordered_map`

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;

// Hauptprogramm.
int main () {
  unordered_map<string, int> tab (1000); // Tabelle.
  string w;                 // Aktuelles Wort.

  // Eingabe zeilenweise lesen und verarbeiten.
  while (getline(cin, 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;
  }
}
```

2.1.7 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.

- Das Feld `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`.

2.1.8 Allgemeines Prinzip

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

- Die Satellitendaten können aber auch fehlen, z. B. wenn das Programm nur die Menge aller verschiedenen Wörter ohne ihre Häufigkeit ausgeben soll.

- 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.

2.1.9 Grundoperationen auf Streuwerttabellen

- Einfügen/Ersetzen eines Objekts x
  1. Wenn die Tabelle bereits ein Objekt x' enthält, das im obigen Sinne gleich x ist, wird es durch x ersetzt.
     (Beachte: x kann andere Satellitendaten als x' besitzen.)
  2. Andernfalls wird x zur Tabelle hinzugefügt, sofern sie noch nicht voll ist.
- Suchen eines Objekts x
  1. Wenn die Tabelle ein Objekt x' enthält, das gleich x ist, wird es zurückgeliefert.
  2. Andernfalls wird das Nichtvorhandensein des Objekts durch einen geeigneten Resultatwert (z. B. `null` in Java) angezeigt.
- Löschen eines Objekts x
  1. Wenn die Tabelle ein Objekt x' enthält, das gleich x ist, wird es aus der Tabelle entfernt.
  2. Andernfalls ist die Operation wirkungslos.

2.1.10 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) eine ganze Zahl zu.
- Inhaltlich gleiche Objekte *müssen* den gleichen Streuwert besitzen (vgl. § 2.1.8). (Das heißt, wenn man in einer Java-Klasse `equals` überschreibt, muss man i. d. R. auch `hashCode` überschreiben.)
- 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 `int`-Werte gibt.

2.2.2 Beispiel 1: Punkte

```
// Punkt im zweidimensionalen Raum.
class Point {
  // Koordinaten des Punkts.
  public final double x, y;

  // Punkt mit Koordinaten x und y konstruieren.
  public Point (double x, double y) {
    this.x = x;
    this.y = y;
  }

  // Zeichenketten-Darstellung des aktuellen Objekts liefern.
  public String toString () {
    return "(" + x + ", " + y + ")";
  }
```

```
  // Vergleich des aktuellen Objekts this (mit Typ Point)
  // mit irgendeinem anderen Objekt other (mit beliebigem
  // dynamischem Typ).
  public boolean equals (Object other) {
    // 1. Wenn other kein Point ist, kann es nicht gleich this sein.
    if (!(other instanceof Point)) return false;

    // 2. Andernfalls kann other in Point that umgewandelt werden.
    Point that = (Point)other;

    // 3. Dann können this und that inhaltlich verglichen werden.
    return this.x == that.x && this.y == that.y;
  }

  // Streuwert für das aktuelle Objekt liefern.
  public int hashCode () {
    // Für Punkte, die gemäß equals gleich sind,
    // erhält man den gleichen Streuwert.
    // return (int)(x + y);
    // Oder besser:
    return (int)x << 16 | (int)y;
  }
}
```

2.2.3 Beispiel 2: Zeichenketten

```
// (Gedachter) Ausschnitt aus der Bibliotheksklasse java.lang.String
public class String {
  private int n;          // Länge der Zeichenkette.
  private char [] s;      // Feld mit den einzelnen Zeichen.

  // Streuwert der Zeichenkette berechnen.
  public int hashCode () {
    // Die einzelnen Zeichen werden quasi als Ziffern einer
    // Zahl h mit Basis 31 interpretiert (obwohl es natürlich
    // mehr als 31 verschiedene Zeichen gibt).
    // Durch arithmetischen Überlauf, der in Java wohldefiniert
    // ist, können auch negative Werte entstehen.
    int h = 0;
    for (int i = 0; i < n; i++) {
      h = h * 31 + s[i];
    }
    return h;
  }

  ......
}
```

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
```

- Problem in Java, C und vielen anderen Programmiersprachen:
  - `h % N` stimmt nur für `h >= 0` (und `N > 0`) garantiert mit der mathematischen Definition von `h mod N` überein.
  - Beispielsweise ist `-14 % 3` in Java gleich `-2` (statt `1`) und liegt damit nicht im verlangten Wertebereich.

- 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 $u = h \cdot A$, $v = u \bmod 1 = u - \lfloor u \rfloor \in [0, 1)$
    und $i = \lfloor v \cdot N \rfloor \in \{0, \ldots, N-1\}$,
    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 Wortlänge des Computers 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 $u' = h \cdot A'$, $v' = u' \bmod 2^w$, $i' = v' \cdot N = v' \cdot 2^p$ und $i = \frac{i'}{2^w} = \frac{v' \cdot 2^p}{2^w} = \frac{v'}{2^{w-p}} \in \{0, \ldots, N-1\}$.
  - Damit sind A', u', v' und i' jeweils um den Faktor 2^w größer als A bzw. u bzw. v bzw. i.
  - $h \cdot A'$ ist prinzipiell eine Zahl mit $2w$ Bits, von der die oberen w Bits bei der praktischen Berechnung mit Wortlänge 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 Java mit Wortlänge $w = 32$ und `s` $= A'$:

```
i = h * s >>> 32 - p = (h * s) >>> (32 - p)
```

- 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:

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

    $v' = u' \bmod 2^{16} = 9707 = 0010010111101011_2$

    $i = \frac{v'}{2^{16-10}} = 151 = 0010010111_2$ (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 (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 Einträge 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 Eintrag 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 ein Feld *tab* der Größe N zur Speicherung verketteter Listen sowie eine Funktion h, die jedem Objekt x einen Index $i \in \{0, \ldots, N-1\}$ zuordnet.
  (Das heißt, h ist eine Streuwertfunktion mit Einschränkung des Wertebereichs.)

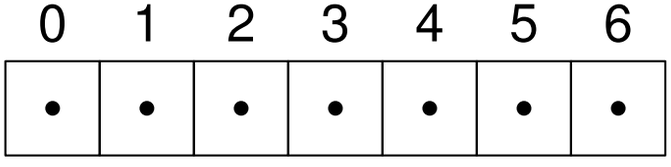
- Einfügen/Ersetzen eines Objekts x
  1. Berechne den Index $i = h(x)$
     und durchsuche die Liste $tab[i]$ nach einem Objekt x', das gleich x ist.
  2. Wenn es ein solches Objekt gibt, ersetze es durch x.
  3. Andernfalls füge x am Anfang der Liste ein.
     (Prinzipiell könnte x irgendwo in die Liste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

- Suchen eines Objekts x
  1. Berechne den Index $i = h(x)$
     und durchsuche die Liste $tab[i]$ nach einem Objekt x', das gleich x ist.
  2. Wenn es ein solches Objekt gibt, liefere es zurück.
  3. Andernfalls liefere $\perp$.

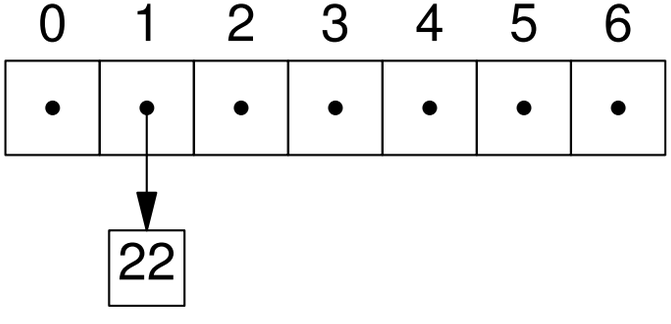
- Löschen eines Objekts x
  1. Berechne den Index $i = h(x)$
     und durchsuche die Liste $tab[i]$ nach einem Objekt x', das gleich x ist.
  2. Wenn es ein solches Objekt gibt, entferne es aus der Liste.

2.5.2 Beispiel

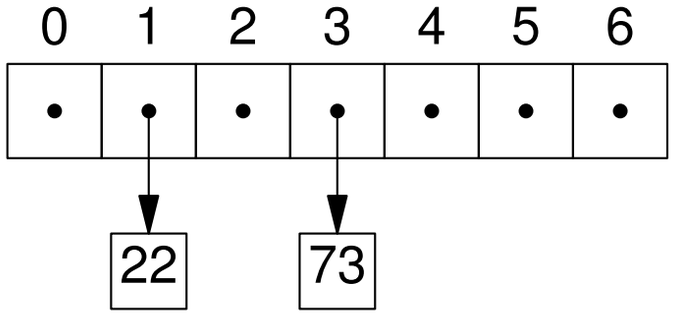
- Sei $N = 7$ und $h(x) = x \bmod 7$ (d. h. die betrachteten Objekte x sind ganze Zahlen).
- Leere Tabelle:



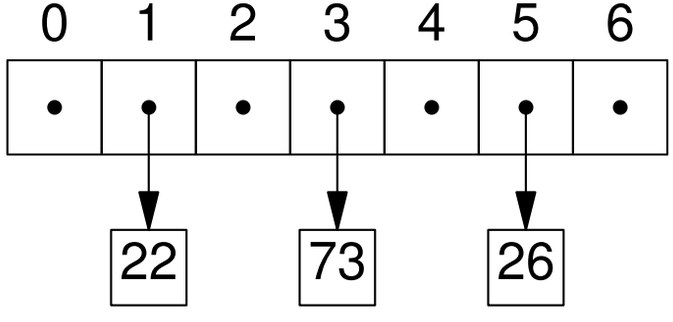
- Einfügen von $x = 22$ mit $h(x) = 22 \bmod 7 = 1$:



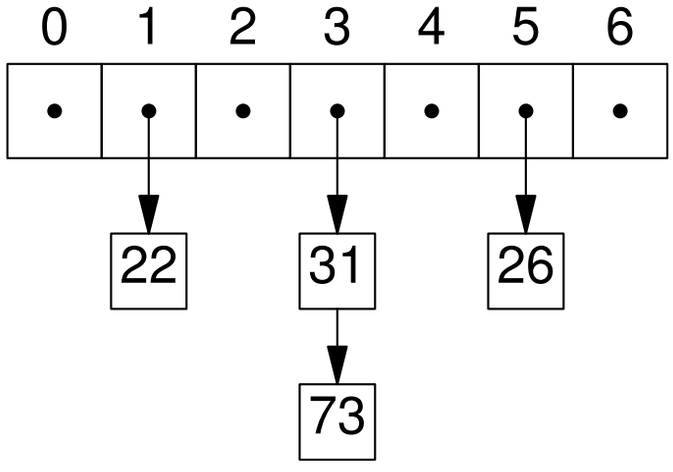
- Einfügen von $x = 73$ mit $h(x) = 73 \bmod 7 = 3$:



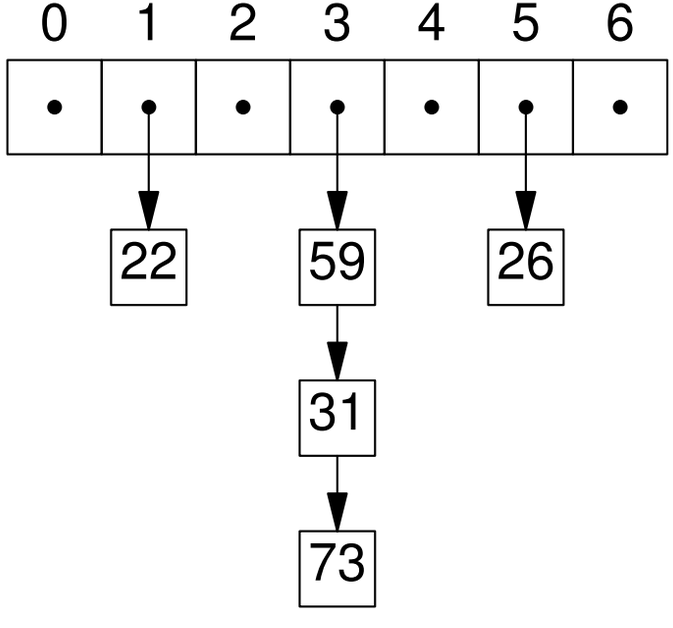
- Einfügen von $x = 26$ mit $h(x) = 26 \bmod 7 = 5$:



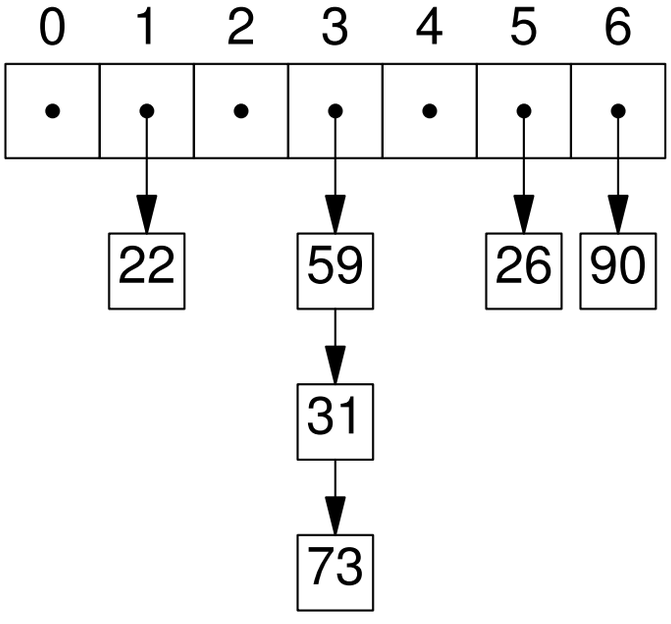
❒ Einfügen von $x = 31$ mit $h(x) = 31 \bmod 7 = 3$:



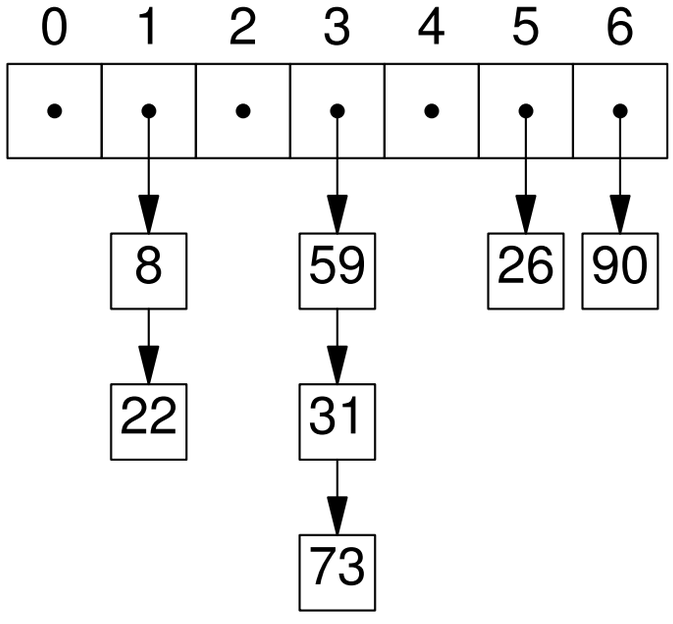
❒ Einfügen von $x = 59$ mit $h(x) = 59 \bmod 7 = 3$:



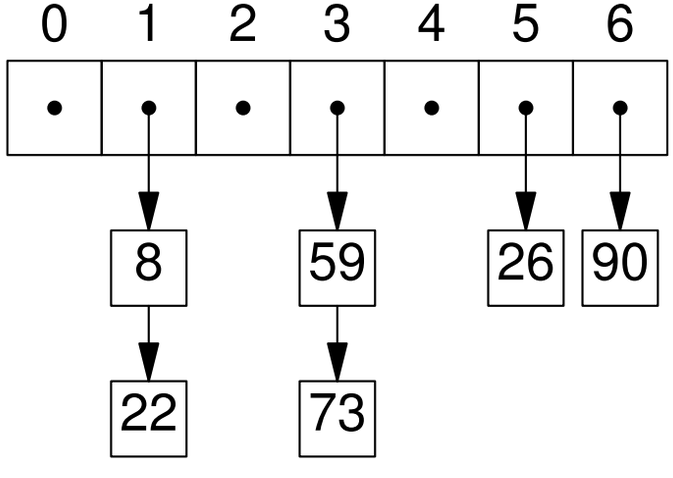
❒ Einfügen von $x = 90$ mit $h(x) = 90 \bmod 7 = 6$:



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



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



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, Suchen und Löschen eines Objekts) wird die Anzahl der erforderlichen Objektvergleiche verwendet.

Schlimmster Fall

- Alle Objekte 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 Chance, 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 Objekt x, das nicht in der Tabelle enthalten ist, werden im Durchschnitt α Objektvergleiche ausgeführt.

Anmerkung:

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

Beweis:

- Sei $i = h(x)$, wobei jeder Index i gleich wahrscheinlich ist.
- Die Länge der Liste $tab[i]$ ist im Durchschnitt α.
- Da x nicht in der Liste enthalten ist, muss diese komplett durchsucht werden.
- Dabei werden im Durchschnitt α Objektvergleiche ausgeführt.

Erfolgreiche Suche

Behauptung:

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

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

Beweis:

- Sei $i = h(x)$.
- Um x zu finden, müssen alle Objekte überprüft werden, die sich in der Liste $tab[i]$ vor x befinden, sowie das Objekt x 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).
- Sei x_j das j-te Objekt, das in die Tabelle eingefügt wurde ($j = 1, \ldots, m$).

- 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 x_j gleich $1 + \frac{m-j}{N}$ ($j = 1, \ldots, m$).
- Die durchschnittliche Laufzeit einer Suche nach irgendeinem 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 werden genauso viele Objektvergleiche durchgeführt wie bei einer erfolglosen Suche nach diesem Objekt.

- Beim Ersetzen eines bereits vorhandenen Objekts werden genauso viele Objektvergleiche durchgeführt wie bei einer erfolgreichen Suche nach diesem Objekt.

Erfolgreiches und erfolgloses Löschen

- Beim Löschen eines vorhandenen Objekts werden genauso viele Objektvergleiche durchgeführt wie bei einer erfolgreichen Suche nach diesem Objekt.

- Beim Löschen eines nicht vorhandenen Objekts werden genauso viele Objektvergleiche durchgeführt wie bei einer erfolglosen Suche nach diesem Objekt.

2.6 Offene Adressierung (open addressing)

2.6.1 Grundoperationen

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

- Hilfsoperation: Suchen des Platzes eines Objekts x
  1. Für $j = 0, \ldots, N-1$:
     1. Berechne den Index $i = s_j(x)$.
     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 gleich x 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 x
  1. Führe die obige Hilfsoperation aus.
  2. Wenn sie einen Index i zurückliefert, speichere x in $tab[i]$.
  3. Andernfalls signalisiere einen Fehler (Tabelle voll).

- Suchen eines Objekts x
  1. Führe die obige Hilfsoperation aus.
  2. Wenn sie „vorhanden“ und einen Index i zurückliefert, liefere das Objekt $tab[i]$ zurück.
  3. Andernfalls liefere $\perp$.

- Löschen eines Objekts x
  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 Platz beim Einfügen wie ein leerer Platz, beim Suchen aber wie ein belegter Platz behandelt wird.

2.6.3 Lineare Sondierung (linear probing)

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

- Definiere $s_j(x) = (h(x) + j) \bmod N$ für $j = 0, \ldots, N-1$,
  d. h. die Sondierungssequenz $s(x)$ eines Objekts x beginnt mit seinem Streuwert $h(x)$ (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: $N = 16$, $h(x) = x$
  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 $s_j(x) = \left(h(x) + \frac{j + j^2}{2} \right) \bmod N$ für $j = 0, \ldots, N-1$,

  d. h. die Sondierungssequenz $s(x)$ eines Objekts x beginnt wieder mit seinem Streuwert $h(x)$ (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:

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

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

- Zu zeigen:
  Die Sondierungssequenz $(s_0(x), \ldots, s_{N-1}(x))$ 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 $j, k \in \{0, \ldots, N-1\}$ mit $j \neq k$ gilt: $s_j(x) \neq s_k(x)$.
  - Annahme: Es gibt $j, k \in \{0, \ldots, N-1\}$ mit $j \neq k$ (o. B. d. A. $j < k$), für die gilt: $s_j(x) = s_k(x)$.
  - Das heißt:

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

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

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

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

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

- Fallunterscheidung:
  - Wenn k und j entweder beide gerade oder beide ungerade sind, ist der Faktor $k + j + 1$ ungerade und somit nicht durch 2 teilbar. Also muss der andere Faktor $k - j$ durch 2^{p+1} teilbar sein. Wegen $j < k$ sowie $k \leq N - 1$ und $j \geq 0$ gilt jedoch: $0 < k - j \leq (N - 1) - 0 = N - 1 = 2^p - 1 < 2^{p+1}$, also kann $k - j$ nicht durch 2^{p+1} teilbar sein.
  - Wenn k gerade und j ungerade ist oder umgekehrt, ist der Faktor $k - j$ ungerade und somit nicht durch 2 teilbar. Also muss der andere Faktor $k + j + 1$ durch 2^{p+1} teilbar sein. Wegen $0 \leq k \leq N - 1$ und $0 \leq j \leq N - 1$ gilt jedoch: $0 < k + j + 1 \leq (N - 1) + (N - 1) + 1 = 2N - 1 = 2^{p+1} - 1 < 2^{p+1}$, also kann $k + j + 1$ 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: $N = 16$, $h(x) = x$
  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 $s_j(x) = (h_1(x) + j\,h_2(x)) \bmod N$ für $j = 0, \ldots, N-1$,
  d. h. die Sondierungssequenz $s(x)$ eines Objekts x beginnt mit dem Streuwert $h_1(x)$ (eingeschränkt auf den Wertebereich $\{0, \ldots, N-1\}$) und durchläuft dann der Reihe nach die Plätze mit Abstand $h_2(x)$, $2\,h_2(x)$, $3\,h_2(x)$, ... von diesem Anfangsplatz.
- Damit ist doppelte Streuung ähnlich zu linearer Sondierung, aber mit einer „Schrittweite“, die vom Objekt x 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: $N = 16$, $h_1(x) = x$, $h_2(x) = x \bmod 15 + 1 - x \bmod 15 \bmod 2$
  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 $(s_0(x), \ldots, s_{N-1}(x))$ 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 $h_2(x)$ und N für alle Objekte x teilerfremd sind:
  - Zu zeigen: Für alle $j, k \in \{0, \ldots, N-1\}$ mit $j \neq k$ gilt: $s_j(x) \neq s_k(x)$.
  - Annahme: Es gibt $j, k \in \{0, \ldots, N-1\}$ mit $j \neq k$, für die gilt: $s_j(x) = s_k(x)$.
  - Das heißt:
    $h_1(x) + j\,h_2(x) \equiv h_1(x) + k\,h_2(x) \pmod{N}$
    $\Leftrightarrow j\,h_2(x) \equiv k\,h_2(x) \pmod{N}$
    $\Leftrightarrow j \equiv k \pmod{N}$ (da $h_2(x)$ und N teilerfremd sind und $h_2(x)$ deshalb ein multiplikatives Inverses modulo N besitzt)
    $\Leftrightarrow j = k$ (da $j, k \in \{0, \ldots, N-1\}$)
  - Widerspruch!

- Anmerkung:
  Damit $h_2(x)$ und N teilerfremd sind, darf $h_2(x)$ 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 $h_2(x)$ und N für alle Objekte x teilerfremd.

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

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

- Praktische Realisierungsmöglichkeit 2:
  - Die Tabellengröße N ist eine Primzahl.
  - Die Zahl N' ist „etwas kleiner“ als N, z. B. $N' = N - 1$.
  - $h_1(x) = x$
  - $h_2(x) = x \bmod N' + 1$
  - Wegen $x \bmod N' \in \{0, \ldots, N' - 1\}$ und $N' \leq N - 1$ ist $h_2(x) \in \{1, \ldots, N'\} \subseteq \{1, \ldots, N - 1\}$.
  - Da N eine Primzahl ist, sind somit alle Werte von $h_2(x)$ teilerfremd zu N.
- Beispiel:
  - $N = 701$, $N' = 700$
- Für $x = 123456$ ergibt sich zum Beispiel:
  - $h_1(x) = 123456 \equiv 80 \pmod{701}$
  - $h_2(x) = 123456 \bmod 700 + 1 = 257$
  - $s_j(x) = (123456 + j \cdot 257) \bmod 701 = (80 + j \cdot 257) \bmod 701$ 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 Objekt x, das nicht in der Tabelle enthalten ist, werden im Durchschnitt höchstens $\frac{\alpha}{1-\alpha}$ Objektvergleiche 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 Objektvergleiche auf jeden Fall nicht größer als $\frac{\alpha}{1-\alpha}$ ist.

Beweis

- Bei der Suche nach dem Objekt x werden so lange Objekte aus der Tabelle mit x 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 Objektvergleiche 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 $0 \leq k \leq m < N$ gilt $\frac{m-k}{N-k} \leq \frac{m}{N}$.
  Aufgrund der Uniform-hashing-Annahme hängt der Ausgang eines Sondierungsversuchs nicht von den vorhergehenden Versuchen ab.)

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

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

$$\sum_{j=1}^{\infty} j \cdot \mathrm{P}(V \geq j) - \sum_{j=1}^{\infty} j \cdot \mathrm{P}(V \geq j + 1) = \sum_{j=1}^{\infty} j \cdot \mathrm{P}(V \geq j) - \sum_{j=2}^{\infty} (j - 1) \cdot \mathrm{P}(V \geq j) =$$

$$1 \cdot \mathrm{P}(V \geq 1) + \sum_{j=2}^{\infty} (j - (j - 1)) \cdot \mathrm{P}(V \geq j) = \mathrm{P}(V \geq 1) + \sum_{j=2}^{\infty} \mathrm{P}(V \geq j) = \sum_{j=1}^{\infty} \mathrm{P}(V \geq j) \leq$$

$$\sum_{j=1}^{\infty} \alpha^j = \frac{\alpha}{1 - \alpha}$$

Einfügen

- Beim Einfügen eines neuen Objekts wird die gleiche Folge von Sondierungsversuchen durchlaufen und damit auch die gleiche Anzahl von Objektvergleichen ausgeführt wie bei einer erfolglosen Suche nach diesem Objekt.

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

Erfolgreiche Suche

Behauptung

- Bei einer Suche nach einem Objekt x, das in der Tabelle enthalten ist, werden im Durchschnitt höchstens $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ Objektvergleiche ausgeführt.

Beweis

- Bei der Suche nach dem Objekt x 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 x, sodass ein Objektvergleich mehr ausgeführt wird.

- Wenn x 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 höchstens $\frac{\frac{j}{N}}{1-\frac{j}{N}} = \frac{j}{N-j}$ Objektvergleiche ausgeführt.

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

- Der Durchschnittswert über alle bis jetzt eingefügten Objekte beträgt:

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

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

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

Ersetzen

- Beim Ersetzen eines bereits vorhandenen Objekts werden genauso viele Objektvergleiche durchgeführt wie bei einer erfolgreichen Suche nach diesem Objekt, d. h. höchstens $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$.

Erfolgreiches und erfolgloses Löschen

- Beim Löschen eines vorhandenen Objekts werden genauso viele Objektvergleiche durchgeführt wie bei einer erfolgreichen Suche nach diesem Objekt.

- Beim Löschen eines nicht vorhandenen Objekts werden genauso viele Objektvergleiche durchgeführt wie bei einer erfolglosen Suche nach diesem Objekt.

2.7 Laufzeitvergleich

- Die folgende Tabelle zeigt die durchschnittliche Anzahl von Objektvergleichen 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 Feldern

- Gegeben sei ein Feld mit Anfangsgröße $N_1 = N$, das sukzessive gefüllt wird.
- Wenn es voll ist, wird es um N Elemente auf $N_2 = 2\,N$ „vergrößert“, indem ein neues Feld der Größe N_2 beschafft und die bereits vorhandenen N_1 Elemente umkopiert werden.
- Wenn dieses Feld wieder voll ist, wird es 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 das Feld 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 das Feld vergrößert wird.

2.8.2 Geometrische Vergrößerung von Feldern

- Gegeben sei ein Feld mit Anfangsgröße $N_0 = N$, das sukzessive gefüllt wird.
- Wenn es voll ist, wird es um den Faktor q auf $N_1 = q\,N$ vergrößert, wobei die bereits vorhandenen N_0 Elemente umkopiert werden müssen.
- Wenn es wieder voll ist, wird es 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 das Feld 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

3 Vorrangwarteschlangen (priority queues)

3.1 Einleitung

- Eine *Maximum-Vorrangwarteschlange* ist eine Datenstruktur, die folgende Operationen möglichst effizient unterstützt:
  - Einfügen eines Objekts mit einer bestimmten Priorität
  - Auslesen und ggf. Entnehmen eines Objekts mit maximaler Priorität
  - Nachträgliches Ändern der Priorität eines Objekts
  - Entfernen eines Objekts
  - Eventuell zusätzlich: Vereinigen zweier Warteschlangen
- Eine *Minimum-Vorrangwarteschlange* unterstützt entsprechend das Auslesen und ggf. Entnehmen eines Objekts mit minimaler statt maximaler Priorität.
- Anwendungsbeispiele:
  - Prozess-Disponent (scheduler) eines Betriebssystems
  - Huffman-Kodierung (vgl. § 4.3)
  - Bestimmte Graphalgorithmen (vgl. § 5.5.3 und § 5.6.5)

3.2 Binäre Halden (binary heaps)

3.2.1 Interpretation eines Felds als Binärbaum

- Element $1 = 1_2$: Wurzelknoten
- Element $2 = 10_2 = 2 \cdot 1 + 0$: Linker Nachfolger des Wurzelknotens
- Element $3 = 11_2 = 2 \cdot 1 + 1$: Rechter Nachfolger des Wurzelknotens
- Element $4 = 100_2 = 2 \cdot 2 + 0$:
  Linker Nachfolger des linken Nachfolgers des Wurzelknotens
- Element $5 = 101_2 = 2 \cdot 2 + 1$:
  Rechter Nachfolger des linken Nachfolgers des Wurzelknotens
- Element $6 = 110_2 = 2 \cdot 3 + 0$:
  Linker Nachfolger des rechten Nachfolgers des Wurzelknotens
- Element $7 = 111_2 = 2 \cdot 3 + 1$:
  Rechter Nachfolger des rechten Nachfolgers des Wurzelknotens
- Usw.

Allgemein

- Zu einem Element i ist
  - Element $2i + 0$ sein linker Nachfolger (falls $2i + 0 \leq m$)
  - Element $2i + 1$ sein rechter Nachfolger (falls $2i + 1 \leq m$)
  - Element $\left\lfloor \frac{i}{2} \right\rfloor$ sein Vorgänger (falls $i > 1$)
- Dabei bezeichnet m die Gesamtzahl der Elemente des Baums, die auch kleiner als die Größe N des Felds sein kann.
- Ebene l ($l = 0, 1, \ldots$) enthält die Elemente 2^l einschließlich bis 2^{l+1} ausschließlich (jedoch bis maximal m einschließlich).
- Damit befindet sich Element i auf Ebene $\lfloor \log_2 i \rfloor$.

Anmerkungen

- Die Multiplikation $2i$ ist effizient als Verschiebung des Bitmusters von i um eine Position nach links (`i << 1` in C, Java, ...) berechenbar.
- Die ganzzahlige Division $\left\lfloor \frac{i}{2} \right\rfloor$ ist effizient als Verschiebung des Bitmusters von i um eine Position nach rechts (`i >> 1` in C, Java, ...) berechenbar.

3.2.2 Minimum- und Maximum-Halden

- Definition:
  Ein Baum erfüllt die *Maximum-Bedingung*, wenn jeder seiner Knoten außer der Wurzel höchstens so groß wie sein Vorgänger ist.

- Folgerung:
  Alle Knoten eines Teilbaums sind höchstens so groß wie die Wurzel dieses Teilbaums.

- Insbesondere:
  Die Wurzel des gesamten Baums ist ein maximales Element des Baums.

- Definition:
  Die ersten *m* Elemente eines Felds stellen eine *binäre Maximum-Halde* dar, wenn der durch diese Elemente definierte Binärbaum die Maximum-Bedingung erfüllt.

- *Minimum-Bedingung* und *Minimum-Halde* sind analog definiert.

3.2.3 Operationen auf Maximum-Halden (auf Minimum-Halden analog)

Absenken eines Elements

- Vorbedingung:
  Linker und rechter Teilbaum (falls vorhanden) des Elements *i* erfüllen die Maximum-Bedingung.

- Nachbedingung/Ziel:
  Der Teilbaum mit Wurzelknoten *i* erfüllt die Maximum-Bedingung.

- Ablauf:

  1 Wenn Knoten *i* keine Nachfolger besitzt, ist nichts zu tun.

  2 Wenn er genau einen (d. h. einen linken) Nachfolger besitzt, ist dies natürlich der größte Nachfolger.

  3 Wenn er zwei Nachfolger besitzt, bestimme den größeren von ihnen.
  (Falls beide gleich sind, wähle willkürlich einen von beiden.)

  4 Wenn Element *i* kleiner als dieser größte Nachfolger ist, vertausche es mit ihm und wiederhole dann die gesamte Operation für diesen Nachfolger.

- Beweis der Korrektheit durch Induktion nach der Tiefe t des betrachteten Teilbaums:
  - Induktionsanfang $t = 0$:
    In diesem Fall tut die Operation nichts, was korrekt ist, weil der Teilbaum bereits die Maximum-Bedingung erfüllt.
  - Induktionsschritt $t \to t + 1$:
    - Wenn Element i kleiner als sein größter Nachfolger ist, wird es mit ihm vertauscht, wodurch die Maximum-Bedingung für den entsprechenden Teilbaum verletzt werden kann.
    - Für die Teilbäume dieses Teilbaums bleibt die Bedingung jedoch erhalten. Daher stellt der rekursive Aufruf der Operation für diesen Teilbaum die Maximum-Bedingung gemäß Induktionsvoraussetzung wieder her.
      (Dieser Teilbaum hat Tiefe $\leq t$.)
    - Aufgrund der zuvor durchgeführten Vertauschung ist die Bedingung dann auch für den Teilbaum mit Wurzelknoten i erfüllt.
    - Andernfalls (Element i nicht kleiner als sein größter Nachfolger) tut die Operation nichts, was korrekt ist:
      Da Element i mindestens so groß wie seine Nachfolger ist und seine Teilbäume die Maximum-Bedingung gemäß Vorbedingung erfüllen, erfüllt auch der Teilbaum mit Wurzelknoten i die Bedingung.

- Die Laufzeit der Operation ist offenbar proportional zur Tiefe t des betrachteten Teilbaums und damit höchstens $O(\log_2 m)$.

Exkurs

- Eine *Schleifeninvariante* ist eine Aussage, die an folgenden Stellen gilt:
  - Vor Beginn einer Schleife
  - Am Anfang und am Ende jedes Schleifendurchlaufs
  - Nach Beendigung der Schleife
- Um zu beweisen, dass eine bestimmte Aussage tatsächlich eine Schleifeninvariante darstellt, genügt es (ähnlich wie bei vollständiger Induktion), zwei Dinge zu zeigen (unter der Annahme, dass die Auswertung der Schleifenbedingung keine Nebeneffekte verursacht):
  - *Initialisierung:*
    Die Invariante gilt vor Beginn der Schleife.
    (Dann gilt sie auch am Anfang des ersten Durchlaufs, sofern es überhaupt einen Durchlauf gibt.)
  - *Aufrechterhaltung:*
    Wenn die Invariante am Anfang eines Durchlaufs gilt, dann gilt sie auch am Ende dieses Durchlaufs (und damit auch am Anfang des nächsten Durchlaufs, sofern es einen solchen gibt).
    (Zusammen mit der Initialisierung folgt dann durch vollständige Induktion, dass die Invariante nach *jedem* Durchlauf gilt.)

- Daraus folgt dann automatisch:
  - *Terminierung:*
    Wenn die Schleife terminiert (was ggf. anderweitig gezeigt werden muss, sofern es nicht offensichtlich ist), dann gilt die Invariante auch nach ihrer Beendigung.

- Beispiel:

```
int pow (int x, int n) {
  int p = 1, k = 0;
  while (k < n) {
    p = p * x;
    k = k + 1;
  }
  return p;
}
```

Um zu zeigen, dass diese Funktion für $n \geq 0$ tatsächlich x^n berechnet, kann die Schleifeninvariante $p = x^k$ verwendet werden:

- Initialisierung:
  Vor Beginn der Schleife gilt: $p = 1 = x^0 = x^k$
- Aufrechterhaltung:
  Wenn die Aussage $p = x^k$ am Anfang eines Durchlaufs gilt, dann gilt am Ende dieses Durchlaufs: $p' = p \cdot x = x^k \cdot x = x^{k+1} = x^{k'}$.
  Dabei bezeichnen p und k die Werte der entsprechenden Variablen am Anfang des Durchlaufs, p' und k' ihre Werte am Ende des Durchlaufs.
- Terminierung:
  Nach Beendigung der Schleife ist $k = n$ (sofern $n \geq 0$ ist) und somit gilt:
  $p = x^k = x^n$.

- Anmerkung:
  for-Schleifen sind für derartige Beweise aus mehreren Gründen schlecht geeignet:
  - Die Veränderung der Laufvariable erfolgt (je nach konkreter syntaktischer Form) mehr oder weniger „versteckt“.
  - Vor Beginn und nach Beendigung der Schleife existiert die Laufvariable meist gar nicht.

  Deshalb sollten for-Schleifen in äquivalente while-Schleifen umgeschrieben werden, wenn die Laufvariable in der Schleifeninvariante gebraucht wird.

Herstellen einer Maximum-Halde

- Gegeben: Ein Feld der Größe N mit beliebigen Elementen.
- Nachbedingung/Ziel:
  Die ersten m Elemente des Felds stellen eine Maximum-Halde dar.
- Ablauf:

  Für $i = \left\lfloor \frac{m}{2} \right\rfloor, \ldots, 1$:

  Führe die zuvor beschriebene Operation „Absenken“ für das Element i aus.
- Äquivalente Formulierung mit while-Schleife:

  1 Setze $i = \left\lfloor \frac{m}{2} \right\rfloor$.

  2 Solange $i \geq 1$ ist:

    1 Führe die zuvor beschriebene Operation „Absenken“ für das Element i aus.

    2 Erniedrige i um 1.

- Beweis der Korrektheit mit Hilfe folgender Schleifeninvariante:
  Alle Teilbäume mit Wurzelknoten $i+1, \ldots, m$ erfüllen die Maximum-Bedingung.
  - Initialisierung:
    - Vor Beginn der Schleife ist $i = \left\lfloor \frac{m}{2} \right\rfloor$.
    - Wegen $\lfloor x \rfloor > x - 1$ gilt für $j = i+1, \ldots, m$:

      $$2j \geq 2(i+1) = 2\left(\left\lfloor \frac{m}{2} \right\rfloor + 1\right) > 2\left(\left(\frac{m}{2} - 1\right) + 1\right) = 2\frac{m}{2} = m.$$

    - Daher besitzen diese Knoten j gemäß § 3.2.1 keine Nachfolger.
    - Also erfüllen die entsprechenden Teilbäume die Maximum-Bedingung.
  - Aufrechterhaltung:
    - Zu Beginn eines Schleifendurchlaufs erfüllen alle Teilbäume mit Wurzelknoten $i+1, \ldots, m$ aufgrund der Invariante die Maximum-Bedingung.
    - Wegen $2i + 0 \geq i + 1$ und $2i + 1 \geq i + 1$ erfüllen insbesondere die Teilbäume des Knotens i die Maximum-Bedingung, sofern sie existieren.
    - Somit ist vor dem Aufruf der Operation „Absenken“ deren Vorbedingung und damit nach ihrem Aufruf auch ihre Nachbedingung erfüllt, d. h. auch der Teilbaum mit Wurzelknoten i erfüllt dann die Maximum-Bedingung.
    - Da am Ende des Schleifendurchlaufs $i' = i - 1$ ist, erfüllen damit alle Teilbäume mit Wurzelknoten $i'+1, \ldots, m$ die Maximum-Bedingung.

- Terminierung:
  - Nach Beendigung der Schleife ist $i = 0$.
  - Aufgrund der Invariante erfüllen also alle Teilbäume mit Wurzelknoten $1, \ldots, m$ die Maximum-Bedingung.
  - Insbesondere erfüllt der gesamte Baum mit Wurzelknoten 1 die Maximum-Bedingung, d. h. die ersten m Elemente des Felds stellen eine Maximum-Halde dar.

Laufzeit

- Die Tiefe des gesamten Baums ist $t = \lfloor \log_2 m \rfloor$.
- Ein Teilbaum, dessen Wurzelknoten sich auf Ebene $l = 0, 1, \ldots$ befindet, hat höchstens Tiefe $t - l$. Dementsprechend ist die Laufzeit der Operation „Absenken“ für einen solchen Teilbaum $O(t - l)$.
- Auf Ebene $l = 0, \ldots, t - 1$ gibt es höchstens 2^l abzusenkende Knoten.
  (Auf der untersten Ebene t werden keine Knoten abgesenkt.)
- Also ist die Gesamtlaufzeit höchstens:

$$\sum_{l=0}^{t-1} 2^l \cdot (t-l) \underset{k=t-l}{=} \sum_{k=1}^{t} 2^{t-k} \cdot k = 2^t \cdot \sum_{k=1}^{t} k \cdot \left(\frac{1}{2}\right)^k \leq m \cdot \sum_{k=1}^{\infty} k \cdot \left(\frac{1}{2}\right)^k = m \cdot \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2m$$

(Beachte: Für $|q| < 1$ gilt: $\sum_{k=1}^{\infty} k \cdot q^k = \frac{q}{(1-q)^2}$.)

Sortieren eines Felds

- Gegeben: Ein Feld der Größe N mit beliebigen Elementen.
- Nachbedingung/Ziel: Die Elemente des Felds sind aufsteigend sortiert.
- Ablauf:
  1. Setze $m = N$
  2. Führe die Operation „Herstellen einer Maximum-Halde“ aus.
  3. Solange $m > 1$ ist:
     1. Vertausche Element 1 und m des Felds.
     2. Verkleinere die Halde um ein Element, d. h. erniedrige m um 1.
     3. Führe die Operation „Absenken“ für Element 1 aus.

- Beweis der Korrektheit mit Hilfe folgender Schleifeninvariante:

  1. Die ersten m Elemente des Felds stellen eine Maximum-Halde dar.
  2. Die übrigen Elemente des Felds sind aufsteigend sortiert und mindestens so groß wie die ersten m Elemente.

  - Initialisierung:
    - Vor Beginn der Schleife stellen die ersten m Elemente eine Maximum-Halde dar.
    - Weitere Elemente gibt es zu diesem Zeitpunkt nicht.
  - Aufrechterhaltung:
    - Aufgrund der Invariante gilt zu Beginn eines Durchlaufs:
      Element 1 ist mindestens so groß wie die Elemente 2, . . ., m
      und höchstens so groß wie die übrigen Elemente des Felds.
    - Also gilt nach dem Vertauschen von Element 1 und m und dem Erniedrigen von m Teil 2 der Invariante.
    - Nach dem Ausführen der Operation „Absenken“ gilt auch Teil 1 der Invariante wieder, der durch das Vertauschen eventuell ungültig geworden war.
  - Terminierung:
    - Nach Beendigung der Schleife ist $m = 1$.
    - Also sind nach Teil 2 der Invariante die Elemente 2, . . ., N aufsteigend sortiert und mindestens so groß wie Element 1.
    - Damit ist das gesamte Feld aufsteigend sortiert.

- Laufzeit
  - Es findet ein Aufruf der Operation „Herstellen einer Maximum-Halde“ mit Laufzeit $O(N)$ sowie $N - 1 = O(N)$ Aufrufe der Operation „Absenken“ statt, deren Laufzeit jeweils höchstens $O(\log N)$ beträgt.
  - Damit ergibt sich eine Gesamtlaufzeit von $O(N) + O(N) \cdot O(\log N) = O(N \log N)$.

3.2.4 Implementierung von Vorrangwarteschlangen

- Eine Maximum-Vorrangwarteschlange kann wie folgt durch eine binäre Maximum-Halde implementiert werden (und analog kann eine Minimum-Vorrangwarteschlange durch eine Minimum-Halde implementiert werden).

- N bezeichne die maximale Kapazität der Warteschlange, d. h. die Größe des zugrundeliegenden Felds.

- m bezeichne die momentane Länge der Warteschlange, d. h. die Anzahl der momentan gespeicherten Objekte.
  Dies ist die gleichzeitig die momentane Größe der Halde.

Hilfsoperationen

- Anheben eines Elements

  Solange das Element einen Vorgänger besitzt und seine Priorität größer als die des Vorgängers ist, vertausche das Element mit seinem Vorgänger.

- Anheben oder Absenken eines Elements

  1 Wenn die Priorität des Elements größer als die seines Vorgängers ist, führe die Operation „Anheben“ für das Element aus.

  2 Andernfalls führe die Operation „Absenken“ für das Element aus (wobei sich die dabei durchgeführten Vergleiche auf die Prioritäten der Elemente beziehen).

Operationen

- Einfügen eines Objekts mit einer bestimmten Priorität

  1 Vergrößere die Halde um ein Element, d. h. erhöhe m um 1,
  und speichere das neue Objekt an Position m.

  2 Führe die Hilfsoperation „Anheben“ für das Element an Position m aus.

- Auslesen eines Objekts mit maximaler Priorität

  Liefere das Objekt an Position 1 der Halde.

- Entnehmen eines Objekts mit maximaler Priorität
  1. Merke das Objekt an Position 1 der Halde, um es später als Resultat liefern zu können.
  2. Versetze das Objekt an Position m der Halde an Position 1 und verkleinere die Halde um ein Element, d. h. erniedrige m um 1.
  3. Führe die Operation „Absenken“ für das Element an Position 1 aus.
  4. Liefere das anfangs gemerkte Objekt zurück.
- Nachträgliches Ändern der Priorität eines Objekts
  1. Ändere die Priorität des Objekts.
  2. Führe die Operation „Anheben oder Absenken“ für das Objekt aus.
- Entfernen eines Objekts
  1. Ersetze das zu entfernende Objekt durch das Objekt an Position m der Halde und verkleinere die Halde um ein Element, d. h. erniedrige m um 1.
  2. Führe die Operation „Anheben oder Absenken“ für das ersetzte Element aus.

Laufzeit der Operationen

- Das Auslesen eines Objekts mit maximaler Priorität benötigt offensichtlich Laufzeit O(1).

- Alle anderen Operationen verschieben ein Objekt innerhalb der Halde ggf. um eine oder mehrere Ebenen nach oben oder unten.
  Dementsprechend ist ihre Laufzeit höchstens so groß wie die Tiefe des Baums, d. h. $O(\log m)$.

3.2.5 Praktisches Problem

- Wie finden die Operationen „Ändern der Priorität“ und „Entfernen“ innerhalb der Halde effizient (d. h. ohne sequentielle Suche) das jeweilige Objekt?
- Erste Idee:
  - Die Operation „Einfügen“ liefert die Position im Feld zurück, an der sie das Objekt gespeichert hat.
  - Die Operationen „Ändern der Priorität“ und „Entfernen“ erhalten diese Position anstelle des Objekts selbst.
- Neues Problem: Die Position eines Objekts ist nicht stabil, weil viele Operationen Objekte in der Halde vertauschen.
- Lösung (wie für viele andere Informatik-Probleme) durch eine zusätzliche Indirektion:
  - Objekte werden nicht direkt in der Halde, sondern in einem zweiten Feld gespeichert.
  - Die Operation „Einfügen“ liefert die Position des Objekts in diesem Feld zurück.
  - In der Halde werden nur diese stabilen Positionen der Objekte gespeichert.
  - Umgekehrt wird im zweiten Feld für jedes Objekt auch seine momentane Position in der Halde gespeichert und bei Bedarf (d. h. wenn sie sich aufgrund einer Vertauschung ändert) aktualisiert.

3.3 Binomial-Halden (binomial heaps)

3.3.1 Binomial-Bäume

Definition

- Ein *Binomial-Baum* mit Grad 0 besteht aus einem einzelnen Knoten.
- Für $k \geq 1$ entsteht ein *Binomial-Baum* mit Grad k aus zwei Binomial-Bäumen mit Grad $k-1$, indem einer von ihnen zu einem Nachfolger des anderen gemacht wird.

Eigenschaften von Binomial-Bäumen

Für jeden Binomial-Baum mit Grad $k \in \mathbb{N}_0$ gilt:

1. Die Tiefe des Baums ist k.
2. Der Grad seines Wurzelknotens ist k.
3. Der Grad aller anderen Knoten ist kleiner als k.
4. Die Nachfolger des Wurzelknotens sind Binomial-Bäume mit Grad $k-1, \ldots, 0$.
5. Der Baum besitzt 2^k Knoten.
6. Auf Ebene l ($l = 0, \ldots, k$) gibt es genau $\binom{k}{l}$ Knoten.

3.3.2 Minimum- und Maximum-Halden

- Definition:
  - Ein *Maximum-Binomial-Baum* ist ein Binomial-Baum, der die Maximum-Bedingung (vgl. § 3.2.2) erfüllt.
  - Eine *Maximum-Binomial-Halde* ist eine (endliche) Folge von Maximum-Binomial-Bäumen, deren Grad streng monoton wächst.
  - *Minimum-Binomial-Baum* und *Minimum-Binomial-Halde* sind analog definiert.
- Folgerung:
  - Alle Binomial-Bäume innerhalb einer Binomial-Halde besitzen unterschiedlichen Grad.
  - Eine Binomial-Halde mit N Elementen besteht aus Binomial-Bäumen mit Grad $k_1 < \ldots < k_p$, sodass gilt: $N = \sum_{i=1}^{p} 2^{k_i}$, d. h. $k_1, \ldots, k_p$ sind genau die Ziffern mit Wert 1 in der Dualdarstellung von N.
  - Sowohl die Anzahl der Bäume in einer solchen Halde als auch deren Grad und Tiefe ist jeweils höchstens $O(\log_2 N)$.

3.3.3 Repräsentation von Binomial-Halden

- Jeder Knoten eines Binomial-Baums bzw. einer Binomial-Halde kann durch eine Datenstruktur mit folgenden Attributen repräsentiert werden:
  - `degree` speichert den Grad des Knotens.
  - `parent` verweist auf den Vorgänger des Knotens.
    Bei einem Wurzelknoten ist `parent` gleich $\perp$.
  - `child` verweist auf den Nachfolger des Knotens mit dem größten Grad.
    Bei einem Blattknoten ist `child` gleich $\perp$.
  - `sibling` verkettet alle Nachfolger eines Knotens in einer nach aufsteigendem Grad sortierten zirkulären Liste, das heißt:
    - Der Nachfolger mit dem größten Grad verweist auf den Nachfolger mit dem kleinsten Grad.
    - Jeder andere Nachfolger verweist auf den Nachfolger mit dem nächstgrößeren Grad.
    - Wenn es nur einen Nachfolger gibt, verweist er auf sich selbst.

    Außerdem verkettet `sibling` die Wurzelknoten aller Bäume der Halde in einer nach aufsteigendem Grad sortierten einfachen Liste, d. h. jeder Wurzelknoten verweist ggf. auf den Wurzelknoten mit dem nächstgrößeren Grad.
  - `entry` verweist auf das Objekt, das in diesem Knoten gespeichert werden soll und das seinerseits auf diesen Knoten zurückverweist (vgl. auch § 3.2.5).

- Die gesamte Halde wird durch einen Verweis auf den Wurzelknoten des Baums mit dem kleinsten Grad repräsentiert. (Bei einer leeren Halde ist dieser Verweis gleich $\perp$.)

Erläuterungen

- Mit den Attributen `child` und `sibling` können ohne Verwendung dynamischer Felder o. ä. für jeden Knoten beliebig viele Nachfolger gespeichert werden.

- Die Gründe für die weiteren Details der Organisation (Reihenfolge der `sibling`-Verkettungen, Verwendung einfacher oder zyklischer Verkettung) werden in § 3.3.5 erläutert, nachdem in § 3.3.4 die Operationen auf dieser Datenstruktur beschrieben wurden.

3.3.4 Implementierung von Vorrangwarteschlangen

- Eine Minimum-Vorrangwarteschlange kann wie folgt durch eine Minimum-Binomial-Halde implementiert werden (und analog kann eine Maximum-Vorrangwarteschlange durch eine Maximum-Binomial-Halde implementiert werden).

Hilfsoperation

- Zusammenfassen zweier Bäume B_1 und B_2 mit Grad k zu einem Baum mit Grad $k + 1$

  1 Wenn die Priorität des Wurzelknotens von B_1 größer als die Priorität des Wurzelknotens von B_2 ist, mache B_1 zum Nachfolger mit dem größten Grad von B_2:

```
B2.sibling = nil
B2.degree = B2.degree + 1
B1.parent = B2
if B2.child == nil then
  B2.child = B1.sibling = B1
else
  B1.sibling = B2.child.sibling
  B2.child = B2.child.sibling = B1
end
```

  2 Andernfalls mache B_2 zum Nachfolger mit dem größten Grad von B_1.

Operationen

- Vereinigen zweier Halden H_1 und H_2 zu einer neuen Halde H

  1. Erstelle einen leeren Zwischenspeicher für bis zu drei Bäume.
  2. Setze $k = 0$.
  3. Solange H_1 oder H_2 oder der Zwischenspeicher nicht leer sind:
     1. Wenn der erste (verbleibende) Baum von H_1 Grad k besitzt, entnimm ihn aus H_1 und füge ihn zum Zwischenspeicher hinzu.
     2. Entsprechend für H_2.
     3. Wenn der Zwischenspeicher jetzt einen oder drei Bäume enthält, entnimm einen von ihnen und füge ihn am Ende von H an.
     4. Wenn der Zwischenspeicher jetzt noch zwei Bäume enthält, fasse sie zu einem Baum mit Grad $k + 1$ zusammen, der als „Übertrag“ für den nächsten Schritt im Zwischenspeicher verbleibt.
     5. Erhöhe k um 1.

  Beispiel: H_1 mit $2^2 + 2^1 + 2^0 = 7$ Elementen und H_2 mit $2^3 + 2^2 + 2^1 = 14$ Elementen ergibt H mit $2^4 + 2^2 + 2^0 = 21$ Elementen

- Einfügen eines Objekts mit einer bestimmten Priorität

  Erzeuge eine temporäre Halde mit einem einzigen Baum mit Grad 0, die das Objekt enthält, und vereinige sie mit der aktuellen Halde.

- Auslesen eines Objekts mit minimaler Priorität

  Suche in der Liste der Wurzelknoten ein Objekt mit minimaler Priorität.

- Entnehmen eines Objekts mit minimaler Priorität

  1 Suche in der Liste der Wurzelknoten ein Objekt mit minimaler Priorität und entferne diesen Knoten aus der Liste.

  2 Wenn dieser Knoten Nachfolger besitzt:
  Vereinige die Liste seiner Nachfolger (beginnend mit dem Nachfolger mit dem kleinsten Grad, der über `child` $\rightarrow$ `sibling` direkt zugreifbar ist)
  mit der verbleibenden Halde.

- Nachträgliches Ändern der Priorität eines Objekts
  1. Wenn die neue Priorität des Objekts kleiner oder gleich der alten ist:
     1. Ändere die Priorität des Objekts.
     2. Solange die Priorität des Objekts kleiner als die seines Vorgängers ist:
        Vertausche die `entry`-Verweise der beiden Knoten
        und aktualisiere die zugehörigen Rückverweise der Objekte auf diese Knoten.
  2. Andernfalls, sofern sich das Objekt nicht in einem Blattknoten befindet:
     Entferne das Objekt und füge es mit der neuen Priorität wieder ein.

- Entfernen eines Objekts
  1. Ändere die Priorität des Objekts quasi auf $-\infty$.
  2. Führe dann die Operation „Entnehmen“ aus.

Laufzeit der Operationen

- Wenn die Halde N Objekte enthält, besitzen alle `sibling`-Listen höchstens Länge $O(\log_2 N)$ und alle Bäume höchstens Tiefe $O(\log_2 N)$ (vgl. § 3.3.2).

- Daher besitzen alle Operationen höchstens Laufzeit $O(\log_2 N)$.

3.3.5 Entwurfsüberlegungen für die Datenstruktur

- Beim Vereinigen zweier Halden müssen ihre Bäume nach aufsteigendem Grad durchlaufen werden.
  Deshalb sind die Wurzelknoten in dieser Reihenfolge verkettet.

- Beim Entnehmen eines Wurzelknotens muss die Liste seiner Nachfolger mit der verbleibenden Wurzelliste vereinigt werden.
  Deshalb ist die Liste der Nachfolger eines Knotens ebenfalls nach aufsteigendem Grad sortiert verkettet.
  Dies weicht von der Darstellung in Cormen et al. (vgl. § 1.3) ab, wo die Liste nach absteigendem Grad verkettet ist und deshalb vor einer Vereinigung erst einmal invertiert werden muss.

- Beim Zusammenfassen zweier Bäume mit dem gleichen Grad wird einer der Bäume zum Nachfolger mit dem größten Grad des anderen Baums, d. h. er muss am Ende der Nachfolgerliste angehängt werden.
  Damit dies möglichst einfach und effizient geht, verweist `child` auf den Nachfolger mit dem größten Grad.

- Damit aber auch der Nachfolger mit dem kleinsten Grad direkt zugreifbar ist, ist die Liste der Nachfolger zirkulär verkettet.

- Für die Wurzelliste genügt jedoch eine einfache Verkettung.

3.4 Vergleich von binären und Binomial-Halden

- Die Laufzeit aller Operationen ist in beiden Implementierungen höchstens $O(\log_2 N)$.
- Bei der Implementierung mit binären Halden ist die Laufzeit der Operation „Auslesen“ sogar nur $O(1)$.
- Die Knoten von Binomial-Halden brauchen relativ viel Platz für Verwaltungsdaten (`degree`, `parent`, `child` und `sibling`), während binäre Halden nur die reinen Nutzdaten speichern.
- Binomial-Halden besitzen aber zwei Vorteile gegenüber binären Halden:
  - Ihre Kapazität ist prinzipiell nicht begrenzt.
  - Zwei Halden können auch effizient vereinigt werden.