Algorithmen und Datenstrukturen

Zuordnungstabellen

Prof. Justus Piater, Ph.D.

12. April 2025

Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

Inhaltsverzeichnis

Einführung

In diesem Kapitel betrachten wir Datenelemente als Schlüssel-Wert-Paare, wobei der Schlüssel dazu dient, den zugehörigen Wert in einer großen Datenmenge möglichst schnell wiederzufinden. Der *Schlüssel* dient also lediglich der *Identifikation* und *Lokalisierung* des Wertes. Abgesehen von Gleichheit definieren wir hier jedoch keine Relation über den Schlüsseln. Wir werden also nicht nach dem Nachfolger eines gegebenen Schlüssels fragen, oder welcher von zwei gegebenen Schlüsseln der kleinere ist.

Ihre Matrikelnummer ist so ein Schlüssel. Das Prüfungsreferat benötigt lediglich Ihre Matrikelnummer, um Ihren Datensatz mit Ihren registrierten Lehrveranstaltungen und Beurteilungen einzusehen. Ein Paketzusteller benötigt lediglich die Tracking-Nummer Ihres Pakets, um den zugehörigen Datensatz mit Zustelladresse und aktuellem Aufenthaltsort des Pakets zu ermitteln.

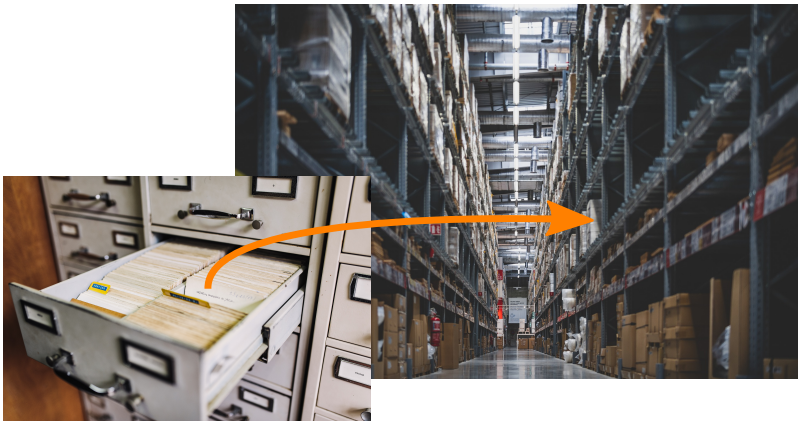
Auch ein URL ist ein solcher Schlüssel: Sie geben ihn in die Adresszeile Ihres Web-Browsers ein, und schon erhalten Sie die Ressource, auf die dieser URL verweist. Daher steht URL schließlich für *uniform resource locator*.

> Dies ist der Kern des abstrakten Datentyps ***Zuordnungstabelle***: der Zugriff auf Datensätze über einen Schlüssel. Zuordnungstabellen werden oft auch als *assoziatives Datenfeld* bezeichnet, und auf englisch als *map*, *dictionary* oder *associative array*.
>
> Zuordnungstabellen können sehr effizient mittels der Datenstruktur *Streuwerttabelle* implementiert werden, besser bekannt als ***Hash-Tabelle***. Hash-Tabellen ermöglichen Zugriff auf einen gegebenen Schlüssel in *konstanter* erwarteter Zeit, also unabhängig von der Anzahl der Elemente, die in der Datenstruktur gespeichert sind. Das ist wirklich erstaunlich! Damit ermöglichen Hash-Tabellen u.a. die praktisch sofortigen Antworten von Internet-Suchmaschinen, trotz der Abermilliarden Datensätze, die sie vorhalten.

1 Abstrakter Datentyp

Video 1 beginnt hier.

Zuordnungstabellen [Slide 1]



[Fotos von Maksym Kaharlytskyi auf Unsplash und von tianya1223 auf pixabay]

Zuordnungstabelle: ***assoziative*** Speicherung von ***Elementen*** – Paaren *einzigartiger Schlüssel* und *Werte* – um auf Daten anhand ihrer Schlüssel zuzugreifen.

Auch bekannt als *assoziatives Datenfeld* oder englisch *map, dictionary, associative array*.

ADT: Zuordnungstabelle *(Map)* [Slide 2]

```
get(k)      // Returns the value v associated with key k,
            // if it exists; otherwise returns null.

put(k, v)   // If M does not have an entry with key k, then
            // adds entry (k, v) to M and returns null.
            // Otherwise, replaces with v the value of the
            // existing entry, and returns the old value.

remove(k)   // Removes from M the existing entry with key k,
            // and returns its value; otherwise returns null.

keySet()    // Returns an iterable collection of all keys.
values()    // Returns an iterable collection of all values
            // (possibly with repetitions).

entrySet()  // Returns an iterable collection of key–value entries.
size()      // Returns the number of entries in M.
isEmpty()   // Returns a boolean indicating whether M is empty.
```

Beispiele:

- Auslieferungslager
- Student ID
- URL
- Benutzernamen

Definition

In einer ***Zuordnungstabelle*** geht es uns um den Zugriff auf einzelne Datensätze über ihren Schlüssel. Nicht mehr und nicht weniger. Darüber hinaus werden keinerlei Annahmen über Daten oder Schlüssel gemacht. Insbesondere sind die Schlüssel in keiner Weise geordnet, weder nach Zahlenwert, noch alphabetisch, noch zeitlich, noch nach Priorität.

Der abstrakte Datentyp Zuordnungstabelle enthält daher lediglich die drei charakteristischen Methoden `get()`, `put()` und `remove()`.

`get(k)` liefert den Wert, also den Datensatz zurück, der unter dem Schlüssel `k` abgelegt wurde, bzw. `null`, falls kein Wert unter dem Schlüssel `k` existiert.

`put(k, v)` speichert den Wert `v` unter dem Schlüssel `k`. Falls bereits ein Wert `w` unter diesem Schlüssel `k` vorhanden ist, wird `w` durch `v` ersetzt und zurückgegeben.

`remove(k)` entfernt das Schlüssel-Wert-Paar `(k, v)` aus der Zuordnungstabelle und liefert `v` zurück, sofern es existiert.

Wie bei allen Containern möchten wir auch über die Elemente einer Zuordnungstabelle iterieren. Diese Elemente bestehen in Schlüssel-Wert-Paaren. Wir können einen Iterator instanziieren, indem wir uns mittels der Methode `entrySet()` einen iterierbaren Container holen und dann dessen Methode `iterator()` aufrufen.

In der Praxis interessieren uns jedoch häufig lediglich die Schlüssel, oder lediglich die Werte, aber nicht die Schlüssel-Wert-Paare. Daher stellt unser abstrakter Datentyp Zuordnungstabelle darüber hinaus die Methoden `keySet()` und `values()` bereit, die uns iterierbare Container liefern, die jeweils die Schlüssel bzw. die Werte der Zuordnungstabelle enthalten.

Die Methode `values()` heißt deshalb nicht etwa `valueSet()`, weil die Werte nicht unbedingt eine Menge darstellen: Eine Menge enthält, per Definition, jeden Wert maximal einmal. In einer Zuordnungstabelle bilden die Schlüssel eine Menge (und damit auch die Schlüssel-Wert-Paare), aber es ist durchaus möglich, dass verschiedene Schlüssel auf den-

selben Wert verweisen. Damit liefert ein Iterator über die Werte einer Zuordnungstabelle möglicherweise mehrfach dieselben Werte.

Solche Wiederholungen von Werten müssen bei einer Iteration nicht unbedingt nacheinander auftreten. Schließlich unterliegen die Schlüssel und damit auch die Werte einer Zuordnungstabelle keiner definierten Ordnung.

Erklärung

Was für Datenstrukturen eignen sich für die Implementierung einer Zuordnungstabelle? Wir könnten beispielsweise unsere n Schlüssel-Wert-Paare in einem sortierten Array ablegen. Dann könnten wir `get()` mittels Binärsuche in einer Zeit von $O(\log n)$ implementieren.

`remove()` muss dann zusätzlich das gefundene Element entfernen und die Lücke schließen, indem es $O(n)$ Einträge im Array verschiebt.

Ähnliches gilt für die Methode `put()`. Sie kann den korrekten Rang für den neuen Schlüssel per Binärsuche finden, und muss dann Platz für das neue Schlüssel-Wert-Paar schaffen.

Ein unsortiertes Array ist noch ineffizienter. Hier müssen `get()` und `remove()` das Array linear durchsuchen. Auch `put()` muss dies tun, weil eine Zuordnungstabelle ja jeden Schlüssel maximal einmal enthalten kann. Damit sind hier alle drei Methoden nicht besser als $O(n)$.

Java-Interface [Slide 3]

```
public interface Map<K,V> {
  int size();
  boolean isEmpty();
  V get(K key);
  V put(K key, V value);
  V remove(K key);
  Iterable<K> keySet();
  Iterable<V> values();
  Iterable<Entry<K,V>> entrySet();
}
```

Beispiel: Wort-Häufigkeiten [Slide 4]

```
import java.io.*;
import java.util.Scanner;
import net.datastructures.Entry;
import net.datastructures.Map;
import net.datastructures.ChainHashMap;

/** A program that counts words in a document, printing the most frequent. */
public class WordCount {
  public static void main() {
    Map<String,Integer> freq = new ChainHashMap<>(); // or any concrete map
    // scan input for words, using all nonletters as delimiters
    Scanner doc = new Scanner(System.in).useDelimiter("[^a-zA-Z]+");
    while (doc.hasNext()) {
      String word = doc.next().toLowerCase(); // convert next word to lowercase
      Integer count = freq.get(word);      // get the previous count for this word
      if (count == null)
        count = 0;                         // if not in map, previous count is zero
      freq.put(word, 1 + count);           // (re)assign new count for this word
    }
    int maxCount = 0;
    String maxWord = "no word";
    for (Entry<String,Integer> ent : freq.entrySet()) // find max-count word
      if (ent.getValue() > maxCount) {
        maxWord = ent.getKey();
        maxCount = ent.getValue();
      }
    System.out.print("The most frequent word is '" + maxWord);
    System.out.println("' with " + maxCount + " occurrences.");
  }
}
```

AbstractMap [Slide 5]

```
/**
 * An abstract base class to ease the implementation of the Map interface.
 *
 * The base class provides three means of support:
 * 1) It provides an isEmpty implementation based upon the abstract size() method.
 * 2) It defines a protected MapEntry class as a concrete implementation of the
 *    entry interface
 * 3) It provides implemenations of the keySet and values methods, based upon use
 *    of a presumed implementation of the entrySet method.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public abstract class AbstractMap<K,V> implements Map<K,V> {

  /**
   * Tests whether the map is empty.
   * @return true if the map is empty, false otherwise
   */
  @Override
  public boolean isEmpty() { return size() == 0; }

  //---------------- nested MapEntry class ----------------
  /**
   * A concrete implementation of the Entry interface to be used
   * within a Map implementation.
   */
  protected static class MapEntry<K,V> implements Entry<K,V> {
    private K k; // key
    private V v; // value

    public MapEntry(K key, V value) {
      k = key;
      v = value;
    }

    // public methods of the Entry interface
    public K getKey() { return k; }
    public V getValue() { return v; }

    // utilities not exposed as part of the Entry interface
    protected void setKey(K key) { k = key; }
    protected V setValue(V value) {
      V old = v;
      v = value;
      return old;
    }

    /** Returns string representation (for debugging only) */
```

```
    public String toString() { return "<" + k + ",␣" + v + ">"; }
  } //------------ end of nested MapEntry class ------------

  // Provides support for keySet() and values() methods, based upon
  // the entrySet() method that must be provided by subclasses

  //------------------ nested KeyIterator class ------------------
  private class KeyIterator implements Iterator<K> {
    private Iterator<Entry<K,V>> entries = entrySet().iterator(); // reuse entrySet
    public boolean hasNext() { return entries.hasNext(); }
    public K next() { return entries.next().getKey(); }    // return key!
    public void remove() { throw new UnsupportedOperationException("remove␣not␣supported"
  } //------------ end of nested KeyIterator class ------------

  //------------------ nested KeyIterable class ------------------
  private class KeyIterable implements Iterable<K> {
    public Iterator<K> iterator() { return new KeyIterator(); }
  } //------------ end of nested KeyIterable class ------------

  /**
   * Returns an iterable collection of the keys contained in the map.
   *
   * @return iterable collection of the map's keys
   */
  @Override
  public Iterable<K> keySet() { return new KeyIterable(); }

  //------------------ nested ValueIterator class ------------------
  private class ValueIterator implements Iterator<V> {
    private Iterator<Entry<K,V>> entries = entrySet().iterator(); // reuse entrySet
    public boolean hasNext() { return entries.hasNext(); }
    public V next() { return entries.next().getValue(); }  // return value!
    public void remove() { throw new UnsupportedOperationException("remove␣not␣supported"
  } //------------ end of nested ValueIterator class ------------

  //------------------ nested ValueIterable class ------------------
  private class ValueIterable implements Iterable<V> {
    public Iterator<V> iterator() { return new ValueIterator(); }
  } //------------ end of nested ValueIterable class ------------

  /**
   * Returns an iterable collection of the values contained in the map.
   * Note that the same value will be given multiple times in the result
   * if it is associated with multiple keys.
   *
   * @return iterable collection of the map's values
   */
  @Override
  public Iterable<V> values() { return new ValueIterable(); }
}
```

DS: Unsortierte Tabelle [Slide 6]

Laufzeiten der Methoden?

```
public class UnsortedTableMap<K,V> extends AbstractMap<K,V> {
  /** Underlying storage for the map of entries. */
  private ArrayList<MapEntry<K,V>> table = new ArrayList<>();

  /** Constructs an initially empty map. */
  public UnsortedTableMap() { }

  // private utility
  /** Returns the index of an entry with equal key, or -1 if none found. */
  private int findIndex(K key) {
    int n = table.size();
    for (int j=0; j < n; j++)
      if (table.get(j).getKey().equals(key))
        return j;
    return -1;                                  // special value denotes that key was not found
  }

  // public methods
  /**
   * Returns the number of entries in the map.
   * @return number of entries in the map
   */
  @Override
  public int size() { return table.size(); }

  /**
   * Returns the value associated with the specified key, or null if no such entry exists.
   * @param key the key whose associated value is to be returned
   * @return the associated value, or null if no such entry exists
   */
  @Override
  public V get(K key) {
    int j = findIndex(key);
    if (j == -1) return null;                   // not found
    return table.get(j).getValue();
  }

  /**
   * Associates the given value with the given key. If an entry with
   * the key was already in the map, this replaced the previous value
   * with the new one and returns the old value. Otherwise, a new
   * entry is added and null is returned.
   * @param key   key with which the specified value is to be associated
   * @param value value to be associated with the specified key
   * @return the previous value associated with the key (or null, if no such entry)
   */
  @Override
  public V put(K key, V value) {
    int j = findIndex(key);
```

```
    if (j == -1) {
      table.add(new MapEntry<>(key, value));    // add new entry
      return null;
    } else                                      // key already exists
      return table.get(j).setValue(value);      // replaced value is returned
  }

  /**
   * Removes the entry with the specified key, if present, and returns its value.
   * Otherwise does nothing and returns null.
   * @param key the key whose entry is to be removed from the map
   * @return the previous value associated with the removed key, or null if no such entry exists
   */
  @Override
  public V remove(K key) {
    int j = findIndex(key);
    int n = size();
    if (j == -1) return null;                   // not found
    V answer = table.get(j).getValue();
    if (j != n - 1)
      table.set(j, table.get(n-1));             // relocate last entry to 'hole' created by remov
    table.remove(n-1);                          // remove last entry of table
    return answer;
  }

  //---------- nested EntryIterator class ----------
  private class EntryIterator implements Iterator<Entry<K,V>> {
    private int j=0;
    public boolean hasNext() { return j < table.size(); }
    public Entry<K,V> next() {
      if (j == table.size()) throw new NoSuchElementException("No further entries");
      return table.get(j++);
    }
    public void remove() { throw new UnsupportedOperationException("remove not supported"
  } //---------- end of nested EntryIterator class ----------

  //---------- nested EntryIterable class ----------
  private class EntryIterable implements Iterable<Entry<K,V>> {
    public Iterator<Entry<K,V>> iterator() { return new EntryIterator(); }
  } //---------- end of nested EntryIterable class ----------

  /**
   * Returns an iterable collection of all key-value entries of the map.
   *
   * @return iterable collection of the map's entries
   */
  @Override
  public Iterable<Entry<K,V>> entrySet() { return new EntryIterable(); }
}
```

Quiz [Slide 7]

Wenn wir eine Zuordnungstabelle auf Basis eines Array implementieren, sollten wir das Array sortiert halten oder nicht?

A: Ja, denn ein sortiertes Array verschafft uns Laufzeitvorteile.

B: Nein, denn ein unsortiertes Array verschafft uns Laufzeitvorteile.

C: Das ist egal; es macht keinen wesentlichen Unterschied.

D: weiß nicht

2 *Lookup Tables* und Hash-Codes

Video 2 beginnt hier.

Lookup Table [Slide 8]

0	1	2	3	4	5	6
		A. Hofer			S. Mair	

Laufzeiten der Methoden `get()`, `put()` und `remove()`?

Unzulänglichkeiten:

- Nur natürliche Zahlen als Schlüssel
  Lösung: ***Hash-Funktion***
- Platzverschwendung im Falle einzelner sehr großer Schlüssel
  Lösung: ***Kompressionsfunktion***

Erklärung

Bei unserer Diskussion sortierter Arrays als Datenstruktur für die Implementierung einer Zuordnungstabelle haben wir das Array als eine Sequenz anonymer Zellen betrachtet. In dieser Sequenz mussten wir unsere Schlüssel suchen und beim Einfügen und Entfernen Schlüssel-Wert-Paare verschieben, was lineare Laufzeiten für zwei unserer drei wichtigsten Methoden nach sich zog.

Wir können unser Array jedoch auch als direkt adressierbare Zellen auffassen, nämlich über ihre Indizes. Falls wir als Schlüssel ausschließlich nicht-negative, ganze Zahlen verwenden, dann können wir im Prinzip diese Schlüssel direkt als Indizes in unser Array verwenden.

Dies ist das Prinzip einer sogenannten ***Lookup Table***. Da ein Schlüssel unmittelbar als Index in das Array fungiert, können wir den zugehörigen Wert in *konstanter* Zeit im Array lokalisieren. Hier finden wir den Datensatz von Frau Mair unter dem Schlüssel 5 in konstanter Zeit am Index 5 im Array. Wir müssen ihn nicht suchen, und wir müssen auch keine Elemente verschieben, denn jedes Feld im Array ist genau einem bestimmten Schlüssel zugeordnet. Damit sind die drei Methoden `get()`, `put()` und `remove()` in konstanter Laufzeit implementierbar.

Lookup Tables sind natürlich höchst attraktiv, weil sie den Zugriff auf Datensätze in konstanter Zeit erlauben, das heißt, in einer Zeit unabhängig von der Anzahl der Datensätze, die in unserer Datenstruktur gespeichert sind. Mit anderen Worten, wir können prinzipiell unbegrenzt große Datenmengen handhaben, ohne dass sich der Zugriff auf einzelne Datensätze verlangsamt. Schneller geht es nicht!

Andererseits sind *Lookup Tables* nur anwendbar, solange ausschließlich ganze Zahlen aus einem begrenzten Intervall als Schlüssel verwendet werden. Das ist natürlich eine große

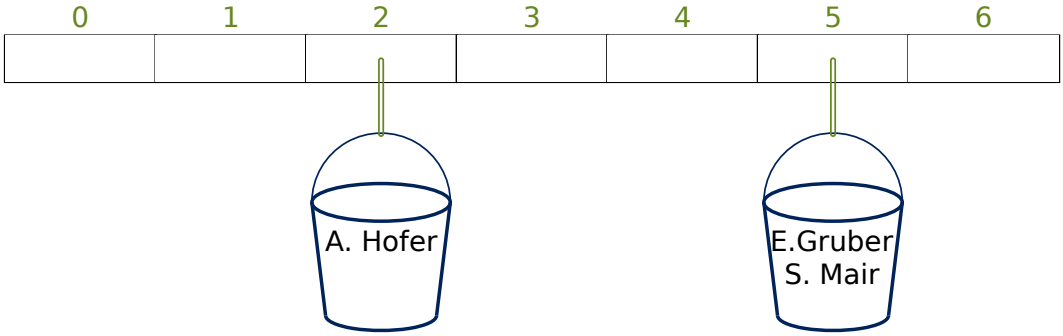
Einschränkung. Denken wir z.B. an eine Kontaktdatenbank: Menschen haben schließlich Namen, nicht Nummern.

Ein zweiter Nachteil ist die Tatsache, das nicht belegte Schlüssel trotzdem Speicherplatz benötigen. Das Array muss schließlich so groß sein, dass der größte verwendete Schlüssel einen gültigen Index darstellt. Dies bedeutet, dass ein einzelner, sehr großer Schlüssel eine gewaltige Verschwendung von Speicherplatz bedeutet, in der Form einer langen Sequenz leerer Zellen.

Können wir diese beiden Probleme beheben, ohne die konstante Laufzeit von `get()`, `put()` und `remove()` aufzugeben? Das ist zu unserem großen Glück möglich. Das Problem großer Schlüssel lösen wir durch eine sogenannte ***Kompressionsfunktion***, die Zahlenwerte aus einem sehr großen Wertebereich auf die begrenzte Größe des Arrays abbildet.

Dies ermöglicht dann auch eine einfache Lösung des ersten Problems. Da jeder Datensatz letztlich durch eine Bitsequenz repräsentiert wird, können wir sogenannte ***Hash-Funktionen*** entwickeln, die aus dieser Zahlenrepräsentation einen geeigneten Zahlenwert berechnen, den wir anschließend durch unsere Kompressionsfunktion schicken können.

DS: Streuwerttabelle *(Hash Table)* [Slide 9]



Erklärung

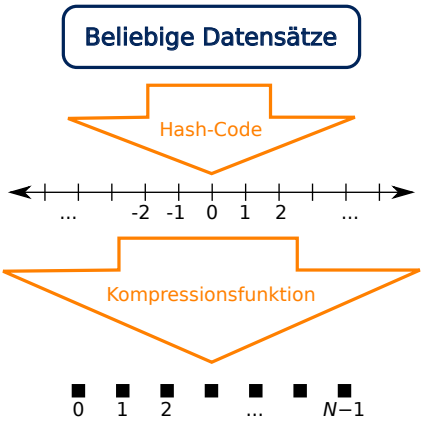
Das Ergebnis ist eine sogenannte ***Streuwerttabelle***, englisch *Hash Table*. Streuwerttabellen sind die bei weitem beliebtesten Datenstrukturen für Zuordnungstabellen. Im Prinzip sind sie nichts weiter als eine *Lookup Table* mit Hash-Funktion und Kompressionsfunktion.

Diese Kombination aus Hash-Funktion und Kompressionsfunktion stellt uns jedoch vor eine offensichtliche Herausforderung: Der berechnete Index ist möglicherweise nicht mehr einzigartig. Es ist nun möglich, dass *verschiedene* Schlüssel auf *denselben* Index im Array abgebildet werden.

Daher sollte man sich eine Streuwerttabelle so vorstellen wie hier illustriert. An jeder Zelle des Arrays hängt ein Eimer, der sämtliche Schlüssel-Wert-Paare enthält, die auf den jeweiligen Index abgebildet werden.

Hash-Funktionen [Slide 10]

- $h(k)$ berechnet den ***Hash-Code*** eines Schlüssels k.
  Ziel: *breitestmögliche Streuung* = Minimierung von ***Kollisionen*** auch ähnlicher Schlüssel.
- Die ***Kompressionsfunktion*** $c(h)$ bildet den Hash-Code auf die Tabellengröße ab.
  Ziel: Minimierung von Kollisionen.
  Diese Trennung vereinfacht die Implementierung mit dynamischen Arrays.
- Element (k, v) wird in $A[c(h(k))]$ gespeichert.



Erklärung

Nun ist es natürlich wichtig, dass verschiedene Schlüssel möglichst auf verschiedene Indizes abgebildet werden. Wenn im Extremfall sämtliche Schlüssel auf denselben Index abgebildet werden, dann haben wir überhaupt nichts gewonnen, und wir müssen einen einzigen, riesigen Eimer nach unserem Schlüssel durchsuchen.

Wenn zwei verschiedene Schlüssel auf denselben Index abgebildet werden, dann handelt es sich um eine sogenannte ***Kollision***. Unser Ziel ist es, Kollisionen zu minimieren. Hierzu müssen wir die Hash-Funktion $h(k)$ und die Kompressionsfunktion $c(h)$ geeignet wählen. Hierin liegt die Wissenschaft von Hash-Tabellen.

Die Hash-Funktion sollte so gewählt werden, dass sie die berechneten Hash-Codes möglichst breit streut. Dies verleiht der Streuwerttabelle ihren Namen. Da in der Praxis die Schlüssel meist sehr ungleichmäßig über den Schlüsselraum verteilt sind, ist es wichtig, dass auch die Hash-Codes einander *ähnlicher* Schlüssel breit gestreut sind, also einander *nicht* ähneln.

Anschließend bildet die Kompressionsfunktion $c()$ den berechneten Hash-Code h auf die Kapazität des Arrays ab, unter Erhaltung einer möglichst gleichmäßigen Streuung der Hash-Codes.

Wenn die Hash-Tabelle voll wird und vergrößert werden muss, dann genügt es, die berechneten Hash-Codes h mittels einer neuen Kompressionsfunktion c auf die neue Tabellengröße abzubilden.

3 Einfache Hash-Codes

Video 3 beginnt hier.

Konvertierung in ganze Zahlen [Slide 11]

Besteht der Schlüssel aus hinreichend wenigen Bits, wird er direkt als ganzzahliger Hash-Code interpretiert.

Besteht er aus zu vielen Bits:

- einige Bits ignorieren (nicht empfehlenswert)
- Segmentweise Addition: $k = (k_0, \ldots, k_{n-1})$, Überläufe ignorierend:

  $$h(k) = \sum_{i=0}^{n-1} k_i$$

  ```
  static int hashCode(long i) {
      return (int) ((i >> 32) + (int) i);
  }
  ```

  Etc., for floats, strings, …

- Dito, aber bitweise XOR statt Addition

Beispiel

Schauen wir uns nun einige verschiedene Hash-Funktionen an. Wir arbeiten immer mit Hash-Codes endlicher Größe, weil es unmöglich ist, Werte in einem unendlichen Wertebereich gleichförmig zu verteilen.

Am einfachsten ist die Situation, wenn der Schlüssel bereits aus einer passenden Anzahl Bits besteht. Dann könnten wir ihn im Prinzip unverändert als ganzzahligen Hash-Code interpretieren. Allerdings werden die Hash-Codes auf diese Weise nicht gestreut.

Besteht der Schlüssel aus mehr Bits als die Hash-Codes, dann besteht ein einfaches Verfahren darin, den Schlüssel segmentweise aufzuaddieren. Überläufe kann man im einfachsten Fall einfach ignorieren. Dabei geht allerdings die Information verloren, die durch die übergelaufenen Bits repräsentiert wurde.

Unser Java-Beispiel berechnet einen 32 Bit breiten Hash-Code aus einem 64-Bit-Integer, indem dessen obere und untere Hälfte addiert werden. Dieses Verfahren lässt sich leicht für beliebig breite Schlüssel verallgemeinern.

Eine beliebte Variante, die recht effektiv für gute Streuung sorgen kann und keine Überläufe verursacht, verwendet bitweises exklusives Oder statt Addition.

Polynomiale Hash-Codes [Slide 12]

Besser, wenn die Reihenfolge signifikant ist:

- `"Algorithmen"`, `"Gehalt␣in␣Rom"`
- `"Datenstruktur"`, `"Durst␣Kater␣tun"`
- `"quid␣est␣veritas"`, `"est␣vir␣qui␣adest"`
  (Die Bibel, Johannes 18,38; Antwort nicht überliefert)

$$h(k) = \sum_{i=0}^{n-1} k_i a^{n-i-1} = (\cdots(((k_0 a + k_1)a) + k_2)a + \cdots + k_{n-2})a + k_{n-1}$$

- Die Multiplikation mit a schafft Platz für die nachfolgenden Segmente.
- Überläufe werden ignoriert (jedoch sollte a niedrigwertige Bits gesetzt haben). Z.B. 8 Bit breite Hash-Codes:

  $01101000 \times 1000 = 01000000$

  $01101000 \times 1001 = 10101000$

- Der Wert von a muss sorgfältig gewählt werden.
  Z.B. 50000 engl. Wörter, $a = 33, 37, 39, 41$: insgesamt weniger als 7 Kollisionen

Erklärung

Wir können einfache Hash-Codes bilden, die Schlüssel segmentweise aufaddieren bzw. mit exklusivem Oder verbinden. Solche einfachen Verfahren haben jedoch einen gravierenden Nachteil: Alle Segmente werden gleich behandelt; ihre Reihenfolge hat keinen Einfluss. Damit kann wichtige Information verloren gehen. Sind die Schlüsselsegmente Buchstaben, dann bedeutet dies, dass alle Texte, die dieselben Buchstaben enthalten, äquivalent sind.

Um die Information der sequenziellen Anordnung zumindest teilweise zu erhalten, können wir die Schlüsselsegmente entsprechend ihrer Sequenz in der Breite über den Hash-Code verteilen. Dies bietet sich insbesondere an, wenn die Segmente deutlich weniger Bits umfassen als der Hash-Code. Ein typisches Beispiel sind Schlüssel, die aus 7-Bit ASCII-Zeichen bestehen, und ein 32 oder 64 Bit breiter Hash-Code.

Ein verbreitetes Verfahren, die Schlüsselsegmente sequenziell über den Hash-Code zu verteilen, besteht darin, jedes Segment k_i entsprechend seines Rangs in der Sequenz mit einer Konstanten zu multiplizieren. Ein sogenannter ***polynomialer*** Hash-Code tut genau dies mit Konstanten der Form a^r für Rang r, und summiert diese Vielfachen der Schlüsselsegmente auf.

Die Formulierung des Polynoms nach dem bekannten Horner-Schema verdeutlicht, wie die gesammelten Schlüsselsegmente bei der Berechnung des Hash-Codes sukzessive nach links verschoben werden, um Platz für das nächste Segment zu schaffen. Segment k_0 wird mit a multipliziert und schafft dadurch Platz zu seiner Rechten für Segment k_1. Dieses wird daraufaddiert. Die Summe wird wiederum mit a multipliziert, und schafft so Platz für k_2, und so weiter.

Das Horner-Schema ist hier nicht nur anschaulich, sondern auch effizient. Es nutzt die Tatsache, dass jede Potenz $a^r = a \cdot a^{r-1}$ ist, und reduziert damit die Zeitkomplexität der Auswertung des Polynoms von $\Theta(n^2)$ auf $\Theta(n)$.

Die sukzessive Multiplikation mit a schafft Platz für nachfolgende Segmente. Hierdurch wird sichergestellt, dass Information erhalten bleibt, die in der Reihenfolge der Segmente liegt. Zu wörtlich sollte man diese Metapher des Platz-Schaffens jedoch nicht nehmen. Denn bei der sukzessiven Multiplikation werden irgendwann Bits links aus dem Hash-Code herausfallen, und wir müssen sicherstellen, dass dadurch nicht die Information der betreffenden, frühen Segmente komplett verloren geht.

Beispiel

In unserem ersten Beispiel mit 8 Bit breiten Hash-Codes wird der Zwischenwert 01101000 mit $a = 1000$, also 8, multipliziert. Dies verschiebt den Zwischenwert um 3 Bits nach links, wodurch die drei höchstwertigen Bits nach links herausgeschoben werden und verloren gehen.

Diesen Verlust können wir vermindern, indem wir a so wählen, dass die Bits bei der Berechnung des Hash-Codes nicht nur nach links verschoben werden, sondern darüber hinaus auch über den rechten Bereich gestreut werden. Das bedeutet, a darf keine Zweierpotenz sein, sondern sollte niederwertige Bits mit Wert 1 enthalten.

In unserem zweiten Beispiel ist $a = 9$. Durch das gesetzte 2^0-Bit wird der Zwischenwert an gleicher Stelle wieder hineinaddiert, wodurch nun nicht nur die zwei, sondern die fünf höchstwertigen Bits Information enthalten.

Wenn der Wert von a sorgfältig gewählt wird, dann kann ein polynomialer Hash-Code sehr effektiv sein. Zum Beispiel führt dieser Hash-Code mit Werten von $a = 33, 37, 39$ oder 41 zu jeweils insgesamt weniger als 7 Kollisionen bei 50 000 englischen Wörtern.

Zyklische Hash-Codes [Slide 13]

Um Überläufe zu vermeiden, kann man die Multiplikationen durch eine zyklische Bit-Verschiebung ersetzen:

```
static int hashCode(String s) {
    int h = 0;
    for (int i = s.length() - 1; i >= 0; i--) {
        // 5-bit cyclic shift of the running sum:
        h = (h << 5) ^ (h >>> 27);
        h += (int) s.charAt(i); // add in next character
    }
    return h;
}
```

Der Betrag der Verschiebung muss sorgfältig gewählt werden.

Erklärung

Um die bei der Berechnung des Hash-Codes links hinausgeschobenen Bits nicht zu verlieren, kann man sie einfach rechts wieder hineinschieben. Diese Methode nennt sich *zyklischer* Hash-Code.

Unser Beispiel verwendet eine solche zyklische Verschiebung um 5 Bits anstatt der Multiplikation mit a, bei einem 32 Bit breiten Hash-Code. Die Verschiebung um 27 Stellen nach rechts platziert dieselben 5 Bits ganz rechts, die bei der Verschiebung um 5 Stellen nach links herausgeschoben werden.

Zyklische Hash-Codes können ebenfalls sehr effektiv sein. Jedoch muss auch hier der Betrag der Verschiebung sorgfältig gewählt werden, um Kollisionen zu minimieren.

[Slide 14]

Bei gut 230000 englischen Wörtern:

Verschiebung in Bits	Kollisionen insgesamt	pro Zelle max.
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

Beispiel

Hier sehen wir eine Aufstellung der Kollisionen, die unter gut 230 000 englischen Wörtern bei zyklischen Bit-Verschiebungen von 0 bis 16 Bits auftreten. Eine Verschiebung um 0 Bits bedeutet, dass die Buchstabencodes einfach aufsummiert werden. Die mittlere Spalte zeigt, dass in diesem Fall praktisch jedes Wort mit mindestens einem anderen Wort kollidiert. Der Minimalwert von 190 Kollisionen insgesamt wird bei einer Verschiebung um 5 Bits erzielt. Deswegen haben wir diese in unserem Beispiel-Algorithmus verwendet.

Die rechte Spalte gibt an, wie viele Wörter maximal auf denselben Hash-Code abgebildet werden. Bei einer Verschiebung um 5 Bits werden maximal 3 verschiedene Schlüssel auf denselben Hash-Code abgebildet. Bei Verschiebungen um 6, 7 oder 12 Bits liegt dieser Wert sogar nur bei 2. Allerdings ist die Gesamtzahl der Kollisionen in diesen Fällen deutlich höher, und damit auch die Zahl der in der Praxis erwarteten Kollisionen bei gleichverteilten Schlüsseln.

Beispiel: Hash-Funktion unserer Animationen [Slide 15]

```
static int hashCode(String s) {
    int h = 0;
    for (int i = s.length() - 1; i >= 0; i--) {
        // Xor 4 lost bits into bits 18–21:
        h = (h << 4) ^ ((h & 0xF0000000) >>> 10);
        h += (int) s.charAt(i); // add in next character
    }
    return h;
}
```

Beispiel

Die Verschiebung um 5 Bits sorgt dafür, dass sich die 7 Bit breiten ASCII-Zeichen beim Hineinaddieren überlappen. Durch die dadurch entstehenden Überträge vermischen sich die Informationen verschiedener Zeichen auf unsystematische Weise. Das sorgt für eine gute Streuung auch ähnlicher Zeichenketten.

Diese Überträge propagieren jedoch nicht unbegrenzt weit nach links. Folglich findet die Streuung im wesentlichen rechts statt. Weiter links im Hash-Code werden die Bits

bei jeder Iteration hauptsächlich ein Stück weitergeschoben, aber ansonsten bleiben sie überwiegend statisch.

Unser zyklischer Hash-Code spielt nun die links herausfallenden Bits ausgerechnet dort wieder ein, wo das Hineinaddieren der folgenden Zeichen ohnehin für gute Streuung sorgt, während weiter links die Bits vor Langeweile einschlafen. Rechts zyklisch hineingeschobene Bits sind also weitgehend verschwendet.

Stattdessen bietet es sich an, diese Bits weiter links einzubringen und dort den stagnierenden Laden wieder etwas aufzumischen. Dies ist der Ansatz des hier gezeigten `hashCode()`-Algorithmus. Hier erfolgt eine Verschiebung um 4 Bits (wo unser voriges Beispiel 5 verwendete), und die überlaufenden 4 Bits werden an den Stellen 18 bis 21 mittels exklusiven Oders wieder in den 32 Bit breiten Hash-Code eingebracht.

Diese Stellen 18 bis 21 liegen recht weit weg vom rechten Bereich des Hash-Codes, wo die neuen Schlüsselsegmente hineinaddiert werden, und lässt diese damit in Ruhe.

Sie liegen in einem Bereich, wo sich viele Schlüsselsegmente durch Addition überlappender Bereiche bereits recht gründlich vermischt haben, aber nun nach etlichen Verschiebungen kaum noch verändert wurden. Genau hier wird nun für neue Abwechslung gesorgt und damit die Streuung der Schlüssel verbreitert.

Gleichzeitig liegt dieser Bereich noch weit genug rechts, dass er nicht sofort wieder nach links hinausgeschoben wird. Dadurch wird die Gefahr reduziert, dass sich am oberen Ende des Hash-Codes ein Zyklus aus Bits bildet, die immer wieder mit sich selbst exklusiv verodert werden.

Dies ist die Hash-Funktion, die in unseren folgenden Animationen Verwendung findet.

Hash-Codes in Java [Slide 16]

- `Object` bietet `hashCode()`.

  Warnung

  Oft eine Funktion der *Adresse* einer Instanz, nicht der *Daten*.

- `String` bietet exzellenten `hashCode()`: 12 Kollisionen bei 230000 englischen Wörtern.

- `Object` bietet `equals()`.

  Warnung

  Möglichst sicherstellen, dass `x.equals(y)` genau dann, wenn `x.hashCode() == y.hashCode()`.

4 Kompressionsfunktionen und Beispiele

Video 4 beginnt hier.

Kompressionsfunktionen [Slide 17]

Eine ***Kompressionsfunktion*** $c(h)$ bildet den Hash-Code $h(k)$ eines Schlüssels k auf das Index-Intervall $[0, N-1]$ des Behälter-Arrays A ab, so dass Kollisionen minimiert werden:

$$P(c(h(k)) = n) = \frac{1}{N}$$

$$P(c(h(k_1)) = c(h(k_2))) = \frac{1}{N}$$

Erklärung

Nachdem wir unseren Hash-Code berechnet haben, der typischerweise 32 oder 64 Bit breit ist, müssen wir seinen Wertebereich auf die begrenzte Anzahl N der Zellen unseres Arrays abbilden. Dadurch können natürlich wiederum Kollisionen entstehen.

Fundamental für das effiziente Funktionieren einer Hash-Tabelle ist, dass die Hash-Codes pseudozufällig möglichst gleichmäßig über das gesamte Array streuen. Damit ist die Wahrscheinlichkeit, dass ein Hash-Code auf eine bestimmte Zelle n abgebildet wird, gleich $1/N$. Die Wahrscheinlichkeit, dass ein *anderer* Hash-Code auf *dieselbe* Zelle n abgebildet wird, ist ebenfalls $1/N$. Damit ist die Wahrscheinlichkeit, dass *beide* verschiedenen Hash-Codes auf diese Zelle n abgebildet werden, gleich $1/N^2$. Da dies für jede der N Zellen n gilt, ist die Wahrscheinlichkeit, dass zwei verschiedene Schlüssel auf *irgendeine* selbe Zelle n abgebildet werden, gleich $N \cdot 1/N^2 = 1/N$.

Eine einfache und effektive Kompressionsfunktion ist der Rest bei Division des Hash-Codes durch N, also $c(h) = h \mod N$. Diese Funktion streut Sequenzen benachbarter Hash-Codes gleichmäßig über das Array.

Beispiel: Hashing und Kompression [Slide 18]

Animation

Sehen wir hier ein paar einfache Beispiele. Wir verwenden die zuvor beschriebene Hash-Funktion, die überlaufende Bits im oberen Bereich des Hash-Codes wieder exklusiv hineinodert, sowie den Divisionsrest durch die Arraygröße N als Kompressionsfunktion. Wir werden kurze Sequenzen von Kleinbuchstaben ans Schlüssel verwenden. Als Wert nehmen wir jeweils die laufende Nummer der `put()`-Operation.

Zuerst rufen wir `put(d,1)` auf. Der Schlüssel `d` besteht nur aus einem einzigen Zeichen. Daher ist sein Hash-Code hier trivialerweise identisch mit seinem ASCII-Code 100. Dieser wird nun modulo $N = 7$ genommen, und das Element wird am resultierenden Index 2 im Array platziert.

Um die Funktionsweise der Hash-Funktion zu verdeutlichen, rufen wir nun `put()` mit einem längeren Schlüssel auf, nämlich `abcdefghi`. Beginnend am Ende der Zeichenkette sehen wir, wie jedes Zeichen sukzessive in den Hash-Code hineinaddiert wird, nachdem der Hash-Code um 4 Bits nach links verschoben wurde.

Diese 4 Bits sind weniger als die Breite von 7 Bit der ASCII-Zeichen. Daher werden die Zeichen durch die Addition vermischt. Auf diese Weise schiebt sich eine zunehmend pseudozufällige Bitsequenz nach links. Da unser Hash-Code 32 Bit breit ist, fallen durch die Verschiebung zunächst lediglich Nullen links heraus.

Diese vier Nullen werden anschließend mit den Bits 18 bis 21 des Hash-Codes exklusiv verodert, was natürlich keinen Effekt hat. So geht es über die Buchstaben `ihgfe` bis `d`.

Nach dem Hineinaddieren des Buchstabens `c` sind die Bits der Schlüsselzeichen inzwischen so weit nach links vorgerückt, dass die bei der Verschiebung herausfallenden 4 Bits Infor-

mation enthalten, nämlich die Werte 0110. Diese sorgen beim anschließenden exklusiven Oder nun dafür, dass Bits 19 und 20 des Hash-Codes von 1 auf 0 gesetzt werden.

Ähnliches geschieht nach dem Einfügen des Buchstabens `b`. Hier haben alle vier links herausgeschobenen Bits den Wert 1, und beim exklusiven Oder wird Bit 19 des Hash-Codes von 1 auf 0 gesetzt.

Abschließend wird noch der Buchstabe `a` hineinaddiert.

Da die Bits der Schlüsselzeichen ganz links angekommen sind, ist der Zahlenwert des Hash-Codes entsprechend groß, nämlich 3 967 265 153. Dieser Wert wird nun modulo 7 genommen und das Element am resultieren Index 5 im Array platziert.

Die Suche nach Elementen mittels `get()` geht ähnlich vonstatten. Wir berechnen den komprimierten Hashcode $c(h(k))$ des gesuchten Schlüssels k, und konsultieren den resultierenden Index.

Im Falle von `get(d)` finden wir den Schlüssel an seinem Index 2. Suchen wir nach dem abwesenden Schlüssel `e`, finden wir seinen Ort mit Index 3 dagegen leer vor.

Wenn wir die Methode `put()` zum wiederholten Mal mit dem Schlüssel `d` aufrufen, dann stellt sie fest, dass bereits ein Element mit diesem Schlüssel existiert. In diesem Fall ersetzt `put(d,3)` den existierenden Wert `1` des alten Elements durch den Wert `3` des neuen Elements.

Kompression durch Division [Slide 19]

$c(h) = h \mod N$

Wie N wählen?

- Codes $\{200, 205, 210, \ldots, 595\}$, $N = 100$: 60 Kollisionen!

  Die 80 Codes verteilen sich auf lediglich 20 verschiedene Zellen, die jeweils 4 codes enthalten. Jeder Code kollidiert also mit drei anderen.

  Warum?

  $d = 5$ teilt $N \Rightarrow$ Vielfache von d treffen wieder aufeinander

- Also wählt man N *prim*.

  Für unsere Codes: keine Kollisionen bei $N = 101$.

- Wiederholte Kollisionen bleiben möglich für Regelmäßigkeiten der Form $pN + qd$.

  Codes 202, 207, 212, ..., 297, *303*, *308*, *313*, ...

- Um N gut zu wählen, müssen wir *die Schlüssel kennen*.

Erklärung

Nehmen wir uns an dieser Stelle einmal kurz Zeit, um einen genaueren Blick auf die Kompressionsfunktion $c(h)$ zu werfen. Sie ist denkbar einfach: Sie bildet lediglich den Divisionsrest des Hash-Codes h dividiert durch die Array-Kapazität N. Dennoch haben wir Spielraum: Wir können die Array-Kapazität frei wählen. Sind alle Arraygrößen äquivalent?

Betrachten wir als motivierendes Beispiel Hash-Codes, die Vielfache einer Zahl d sind, hier $d = 5$. Solche Konstellationen können z.B. durch unglücklich gewählte Hash-Funktionen entstehen. Wenn wir nun $N = 100$ wählen, dann kollidiert 200 mit 300, 400 und 500. 205 kollidiert mit 305, 405, 505, und so weiter. Jeder Hash-Code kollidiert also mit 3 anderen, macht insgesamt 60 Kollisionen, wobei 80 Felder unseres Arrays unbelegt bleiben!

Woran liegt das, und wie können wir die Ursache beseitigen?

Durch den Modulo-Operator kollidieren Hash-Codes, die Vielfache von N sind. Unsere Hash-Codes sind Vielfache von 5. Da $N = 100$ ein Vielfaches von 5 ist, kollidieren unsere Hash-Codes also mit schöner Regelmäßigkeit.

Das Problem ist also, dass d N teilt. Wir können uns sehr einfach davor schützen, dass eine solche regelmäßige Struktur von Vielfachen zu Kollisionen führt: Wir müssen einfach dafür sorgen, dass d N *nicht* teilt, egal, welchen Wert d hat. N sollte also so wenige Teiler wie möglich haben. Folglich setzen wir N auf eine Primzahl.

In unserem Beispiel bedeutet dies: Wenn wir die Array-Kapazität von $N = 100$ auf die Primzahl $N = 101$ erhöhen, dann erleiden unsere 80 Hash-Codes *überhaupt keine* Kollisionen.

Natürlich können wir für jede Kompressionsfunktion böswillige Regelmäßigkeiten in Hash-Codes konstruieren, um Kollisionen zu provozieren. Hier erzielen wir dieselben Kollisionen für Hash-Codes der Form $pN + qd$, also Vielfache von N plus Vielfache eines fixen Offsets d. Für $p \cdot 101 + q \cdot 5$ haben wir dann Hash-Codes 202, 207, 212, usw., die mit den Hash-Codes 303, 308 bzw. 313 kollidieren, und so weiter. Statt 200, 300, 400 und 500 kollidieren dann also 202, 303, 404 und 505.

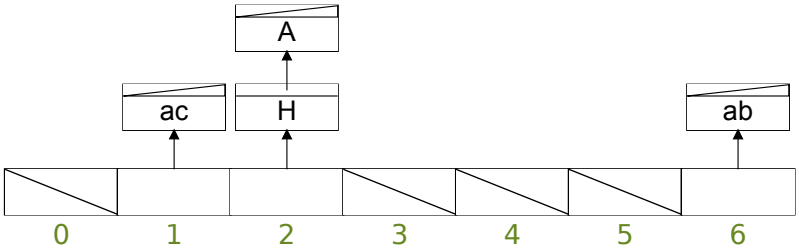
Letztlich hängt also die optimale Wahl der Hash- und Kompressionsfunktionen von unserer Kenntnis der Schlüssel ab.

5 Kollisionsbehandlung: Überblick

Video 5 beginnt hier.

Kollisionsbehandlung durch *externe Verkettung* [Slide 20]

Jeder Behälter $A[i]$ speichert alle Elemente mit $c(h(k_j)) = i$.



Laufzeiten der Methoden im besten, schlechtesten und erwarteten Fall?

Mit einer *guten* Hash-Funktion h ist die erwartete Größe eines Buckets $n \,/\, N$; somit können wir eine Zugriffszeit auf einen bestimmten Schlüssel von $O(\lceil n/N \rceil)$ erwarten. Dies ist $O(1)$ falls $n \in O(N)$.

Q: Können wir die asymptotischen Laufzeiten der Methoden weiter reduzieren, indem wir statt der unsortierten Listen eine andere Datenstruktur verwenden, die sublineare Suchzeiten ermöglicht?

A: ja; **B:** nein; **D:** weiß nicht

Erklärung

Wir müssen einsehen, dass wir Kollisionen im Allgemeinen nicht komplett vermeiden können. Was tun wir nun im Falle einer Kollision?

Eine nahe liegende Lösung ist, alle Elemente, die auf denselben Index im Array hashen, dort in einer geeigneten Container-Datenstruktur abzulegen. Nun enthält die Hash-Tabelle also nicht die Elemente selbst bzw. Zeiger auf diese, sondern jede Zelle enthält eine Referenz auf ihren Container, der seinerseits die gehashten Elemente enthält.

Diese Art der Kollisionsbehandlung nennt man ***externe Verkettung***.

Als Container-Datenstruktur werden üblicherweise verkettete Listen verwendet, aber andere Datenstrukturen wie z.B. eine ArrayList bieten sich ebenso an.

Animation

In unserer Animation führen wir als erstes `put(d,1)` aus, und anschließend `put(k,2)`. Hier kollidiert der Schlüssel `k` mit dem Schlüssel `d`.

Bevor das neue Element mit Schlüssel `k` eingefügt werden kann, muss noch die komplette verkettete Liste am Index 2 abgesucht werden, ob sie nicht bereits ein Element mit diesem Schlüssel enthält. Schließlich dürfen Zuordnungstabellen jeden Schlüssel nur einmal enthalten.

Nebenbei bemerken wir an dieser Stelle, dass unsere Hash-Funktion benachbarte Buchstaben des Alphabets auf benachbarte Zellen abbildet. Dies trifft jedoch nur auf Schlüssel zu, die aus einem einzigen Zeichen bestehen. Allgemein wäre ein solches Verhalten sehr problematisch, weil Schlüssel einander in der Praxis oft ähneln und sich dadurch Cluster in der Hash-Tabelle bilden würden.

Um zu zeigen, dass längere benachbarte Schlüssel tatsächlich gestreut werden, fügen wir hier noch Elemente mit den beiden benachbarten Schlüsseln `aa` und `ab` ein. Wie gewünscht, werden diese beiden Schlüssel in nicht benachbarten Zellen platziert.

Wenn wir nun unser Element mit dem Schlüssel `d` mittels `get(d)` *suchen*, dann wird wieder zunächst der Hash-Code des Schlüssels `d` berechnet, um seinen Index im Array zu ermitteln. Nun müssen wir noch über die dort angehängte verkettete Liste iterieren, um das gesuchte Element zu finden bzw. festzustellen, dass es nicht existiert.

Zu guter Letzt entfernen wir noch das Element mit dem Schlüssel `k` durch den Aufruf `remove(k)`.

Erklärung

Was sind die asymptotischen Laufzeiten von `get()`, `put()` und `remove()`? Jede dieser Methoden muss als erstes den Hash-Code des betroffenen Schlüssels berechnen. Unsere Hash-Funktion benötigt dazu lineare Zeit in der Länge des Schlüssels. Ist die Länge der verwendeten Schlüssel jedoch begrenzt, dann ist die Berechnung des Hash-Codes $O(1)$.

Anschließend muss die am ermittelten Index hängende verkettete Liste nach dem gefragten Schlüssel durchsucht werden. Dies beansprucht lineare Zeit in der Länge dieser Liste.

Im besten Fall enthält diese Liste kein oder nur ein Element. Damit ist die Gesamtlaufzeit $O(1)$, denn alle weiteren Operationen wie Einfügen oder Entfernen eines Elements lassen sich in konstanter Laufzeit implementieren.

Im schlechtesten Fall, der hoffentlich sehr selten oder nie auftritt, kollidieren sämtliche Schlüssel der n Elemente, die die Hash-Tabelle aktuell speichert. Damit degeneriert die Hash-Tabelle zu einer unsortierten, verketteten Liste, und die Laufzeiten aller drei Methoden sind $O(n)$.

Wie sieht es mit dem wichtigen Erwartungsfall aus? Eine gute Hash-Funktion verteilt die Schlüssel gleichmäßig über die Hash-Tabelle. Die Wahrscheinlichkeit, eine bestimmte Zelle zu treffen, beträgt dann $1/N$, und die erwartete Länge jeder der verketteten Listen ist n/N. Damit ist die erwartete Zugriffszeit auf einen gegebenen Schlüssel $O(n/N)$.

Dies bedeutet, dass wir eine erwartete Laufzeit von $O(1)$ sicherstellen können, indem wir N proportional zu n wählen!

Wir können also tatsächlich konstante erwartete Zugriffszeiten erzielen, unabhängig von der Anzahl n der Datensätze! Das bedeutet, dass wir quasi unbegrenzt große Datenmengen verwalten können!

Dies gibt Ihnen einen kleinen Teil einer Vorstellung davon, wie z.B. große Internet-Suchmaschinen Antworten im Bruchteil einer Sekunde liefern können, trotz der Abermilliarden Datensätze, die sie vorhalten.

Kollisionsbehandlung durch *offene Adressierung*: Lineares Sondieren [Slide 21]

Erklärung

Externe Verkettung hat den Nachteil, dass externe Datenstrukturen angelegt werden müssen, mit den damit eingehenden Problemen wie Code-Komplexität, Speicher-Fragmentierung, und schlechteren Lokalitätseigenschaften. Es liegt die Idee nahe, kollidierende Elemente einfach anderswo im Hash-Array unterzubringen, wo noch Platz ist. Wird der Platz knapp, müssen wir das Array ja ohnehin vergrößern.

Diese Art der Kollisionsbehandlung nennt sich ***offene Adressierung***.

Animation

Unsere Animation zeigt die einfachste Variante der offenen Adressierung, das ***lineare Sondieren***. Hier fügen wir wieder ein Element mit dem Schlüssel `k` ein, der mit dem bereits existierenden Schlüssel `d` kollidiert. Beim linearen Sondieren gehen wir einfach so lange im Array weiter, bis ein freies Feld gefunden ist, und legen dort den neuen Schlüssel ab.

Ebenso verfahren wir mit dem Schlüssel `r`, der ebenfalls mit `d` und `k` kollidiert. Dieses Element findet also erst *zwei* Felder weiter seinen Platz.

Entfernen wir nun das Element mit dem Schlüssel `k`. Wenn wir danach z.B. das Element mit dem Schlüssel `r` suchen, dann beginnt die Suche beim Index 2. Dieses ist durch einen anderen Schlüssel `d` belegt; also geht die Suche im Nachbarfeld Nr. 3 weiter. Dieses ist leer! Normalerweise würden wir deshalb dort die Suche erfolglos abbrechen. Aber dann würden wir den tatsächlich vorhandenen Schlüssel `r` übersehen!

Wie können wir das vermeiden? Einfach das gesamte Array abzusuchen kommt nicht in Frage, denn damit wäre unsere Laufzeit mit $O(N)$ katastrophal.

Eine Lösung besteht darin, Felder, deren Element entfernt wurde, als *gelöscht* zu markieren. Trifft die Suche eines Schlüssels auf ein gelöschtes Feld, dann bricht sie nicht ab, sondern wird fortgesetzt. Die Suche wird erst beendet, wenn sie auf ein *leeres* Feld trifft.

In unserem Beispiel ist das Feld Nr. 3 in der Tat als gelöscht markiert. Daher wird das Element mit dem Schlüssel `r` gefunden, obwohl es nicht an seinem eigenen Index liegt und ein Vorgängerelement gelöscht wurde.

Fügen wir nun ein weiteres Element mit dem Schlüssel `e` ein. Es findet sein Feld mit Index 3 als gelöscht markiert vor, und kann sich dort also entspannt niederlassen. Vorher muss es jedoch, wie immer, sicherstellen, dass nicht bereits ein Element mit dem Schlüssel `e` existiert.

6 Kollisionsbehandlung: Offene Addressierung

Video 6 beginnt hier.

Kollisionsbehandlung durch *offene Adressierung* [Slide 22]

Vermeidet externe Datenstrukturen.

`get()`: Suche k in $A[(c(h(k)) + f(i)) \mod N]$ für $i = 0, 1, ...,$ bis

- der Schlüssel k gefunden ist,
- ein *leeres* Feld gefunden ist, oder
- das Array komplett abgesucht wurde.

`put()`:

- Ist `get()` erfolgreich, ersetze v.
- Andernfalls nutze das erste *gelöschte* oder *leere* Feld, das `get()` gefunden hat.

`remove()`: Markiere das Feld *gelöscht*.

Anmerkung

Felder können nicht auf *leer* zurückgesetzt werden. Daher müssen die Elemente periodisch in ein neues Array gehasht werden.

Es gibt andere $O(1)$ `remove()`-Algorithmen, die ohne *gelöscht*-Markierungen auskommen.

Beispiel:

$$\begin{aligned} c(h(k)) &= k \mod 5 \\ f(i) &= i \end{aligned}$$

```
put( 5, 'five')
put(10, 'ten')
put(15, 'fifteen')
remove(10)
get(15)
```

Erklärung

Stellen wir hier noch einmal die Funktionsweise der offenen Adressierung im Detail zusammen. Der Hauptpunkt ist, dass wir nicht mehr einfach von *freien* Feldern reden, sondern zwischen *leeren* und *gelöschten* Feldern unterscheiden.

Ein Aufruf von `get(k)` sucht nach dem Schlüssel `k` in einer Sequenz von Feldern, bis entweder der Schlüssel `k` gefunden ist, ein *leeres* Feld angetroffen wurde, oder das Array komplett abgesucht wurde. Diese Sequenz wird dadurch erzeugt, dass bei jedem Schritt zum komprimierten Hash-Code ein Wert $f(i)$ addiert wird, modulo N, wobei sich i bei jedem Schritt um 1 erhöht.

Bei der linearen Sondierung ist $f(i) = i$, d.h. die Felder dieser Sequenz liegen nebeneinander.

Die Methode `put()` platziert ein neues Element undifferenziert am ersten *leeren* oder *gelöschten* Feld.

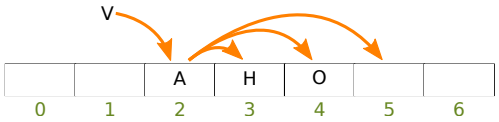
`remove()` markiert das frei gewordene Feld als *gelöscht*.

Beachten Sie, dass Felder niemals auf *leer* zurückgesetzt werden. Damit sind irgendwann alle Felder der Hash-Tabelle entweder belegt oder gelöscht, und jede Suche muss das gesamte Array absuchen. Um dies zu vermeiden, müssen die Elemente hin und wieder in ein neues, komplett *leeres* Array gehasht werden. Es gibt Methoden, die dies vermeiden und komplett ohne *gelöscht*-Markierungen auskommen, aber die sind etwas komplizierter.

Offene Adressierung: Varianten [Slide 23]

Lineares Sondieren:

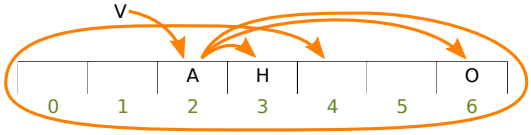
$f(i) = i$



Nachteil: Kollidierende Elemente belegen zusammenhängende Sequenzen in der Tabelle (***clustering***), was zur Degeneration der Zugriffsoperationen führt.

Quadratisches Sondieren:

$f(i) = i^2$



Nachteile:

- Kollidierende Elemente belegen Muster in der Tabelle (***secondary clustering***), was zur Degeneration der Zugriffsoperationen führt.
- Für $n \geq \frac{N}{2}$ ist nicht mehr garantiert, dass ein leeres bzw. gelöschtes Feld auch gefunden wird.

Doppeltes Hashing:

$f(i) = ih'(k)$

Üblich zum Beispiel $h'(k) = q - (k \mod q)$ mit Primzahl $q < N$.

Vorteil: Schlüssel, die unter h kollidieren, kollideren meist nicht auch unter h', also keine Clusterbildung.

Pseudozufälliges Sondieren: $f(i)$ basiert auf einer pseudo-zufälligen Zahlenfolge.

Erklärung

Betrachten wir nun die wichtigsten Varianten der offenen Adressierung. Das ***lineare Sondieren*** haben wir bereits kennengelernt. Beim i-ten Schritt wird i zum komprimierten Hashcode addiert, modulo N. Dadurch wird bei jedem Schritt das folgende Feld konsultiert.

Lineares Sondieren neigt zur Clusterbildung im Hash-Array. Nicht nur werden kollidierene Hash-Codes zusammenhängend angeordnet. Die umgeleiteten Elemente verbannen ihrerseits benachbarte Elemente von ihrem angestammten Platz, was das Problem noch verschlimmert.

Das ***quadratische Sondieren*** funktioniert genauso wie das lineare Sondieren, nur dass die Schrittgrößen *quadratisch* wachsen. Bei $i = 1$ wird also im folgenden Feld gesucht, bei

$i = 2$ 4 Felder weiter, bei $i = 3$ 9 Felder weiter, und so fort, jeweils modulo N.

Durch diese wachsenden Schrittgrößen werden die Elemente breiter gestreut. Allerdings tendiert auch das quadratische Sondieren zur Bildung gewisser Muster in der Arraybelegung, das sogenannte *secondary clustering*.

Linearem und quadratischem Clustering ist gemeinsam, dass kollidierende Schlüssel immer wieder demselben Pfad folgen. Dies ist eine Ursache für die Clusterbildung und führt zu relativ langen Suchpfaden.

Das sogenannte ***doppelte Hashing*** vermeidet diese Pfadwiederholungen, indem i mit dem Wert einer zweiten Hash-Funktion multipliziert wird. Damit folgen kollidierende Schlüssel *verschiedenen* Suchpfaden, denn es ist extrem unwahrscheinlich, dass zwei verschiedene Schlüssel unter *beiden* Hash-Funktionen kollidieren.

Kollisionsbehandlung durch *offene Adressierung*: Doppeltes Hashing [Slide 24]

Animation

In unserem Beispiel wird wieder ein Element mit Schlüssel `d` auf den Index 2 gehasht. Der Schlüssel `k` des nächsten Elements kollidiert mit `d`. Daher wird nun ein zweiter Hash-Code mit der ***sekundären*** Hash-Funktion 5 minus ASCII-Code modulo 5 berechnet und zum ursprünglichen Index addiert. Hier ist der sekundäre Hash-Code 3 und damit der neue Index $2 + 3 = 5$. Das Element wird nun in diesem, freien Feld abgelegt.

Als nächstes fügen wir einen Schlüssel `r` ein, der ebenfalls mit `d` kollidiert. Dessen sekundärer Hash-Code ist jedoch 1 und kollidiert nicht mit dem sekundären Hash-Code des Schlüssels `k`.

Fies und gemein, wie ich bin, habe ich den nächsten Schlüssel `au` so gewählt, dass sein primärer Hash-Code nicht nur mit `d`, `k` und `r` kollidiert, sondern darüber hinaus kollidiert sein sekundärer Hash-Code von 1 mit `r`. Daher bleibt die Sondierung mit $i = 1$ erfolglos, und erst beim nächsten Schritt, mit $i = 2$, findet sich ein freies Feld am Index $2 + 2 \cdot 1 = 4$ modulo 7 gleich 4.

7 Hash-Tabellen: Wichtige Aspekte

Video 7 beginnt hier.

Universelle Hash-Funktionen [Slide 25]

Für eine gegebene Hash-Funktion kann ein Angreifer kollidierende Schlüssel generieren.

Einziger Ausweg: $h(\cdot)$ *zufällig* wählen.

Definition: Sei $\mathcal{H}$ eine endliche Menge von (komprimierenden) Hash-Funktionen $h : U \to [0, N-1]$. $\mathcal{H}$ ist ***universell***, falls für jede zwei ungleichen Schlüssel $k, l \in U$ die Anzahl der Funktionen $h \in \mathcal{H}$ mit $h(k) = h(l)$ nicht größer ist als $|\mathcal{H}|/N$.

Anders gesagt, mit einer zufällig gewählten Hash-Funktion kollidieren k und l mit einer Wahrscheinlichkeit von $1 \,/\, N$.

Erklärung

Wir haben mehrfach gesehen, dass man Kollisionen provozieren kann, wenn man die Hash-Funktion kennt. Hier tobt sich nicht nur Murphys Gesetz gerne aus, sondern es bietet auch Angreifern Möglichkeiten für *Denial-of-Service*-Attacken.

Der einzige Ausweg ist, Hash-Funktionen *zufällig* zu erzeugen, und zwar so, dass zwei beliebige, verschiedene Schlüssel insgesamt mit einer Wahrscheinlichkeit von $1/N$ kollidieren, über viele zufällig gewählte Hash-Funktionen hinweg.

Mit anderen Worten, nehmen wir an, wir wählen zufällig zwei verschiedene Schlüssel k und l. Im besten Fall wird der zweite Schlüssel mit einer Wahrscheinlichkeit von $1/N$ mit dem ersten kollidieren. Mit einer gegebenen Hash-Funktion mag diese Kollision eintreten, mit einer anderen nicht. Unser Ziel ist nun, dass diese geringst-mögliche Kollisionswahrscheinlichkeit von $1/N$ allgemein erreicht wird, mit einer Hash-Funktion, die wir *zufällig* aus einer Menge $\mathcal{H}$ gegebener Hash-Funkionen auswählen. Anders gesagt, allenfalls jede N-te Hash-Funktion aus $\mathcal{H}$ soll zu einer Kollision von k und l führen.

Ist dies der Fall für jede beliebigen zwei verschiedenen Schlüssel, dann ist diese Menge $\mathcal{H}$ *universell*.

Unter jeder Hash-Funktion *müssen* irgendwelche Schlüssel kollidieren. Universelle Hash-Funktionen stellen sicher, dass es nicht immer dieselben Schlüssel sind, die unter vielen Hash-Funktionen kollidieren. Das Risiko von Kollisionen wird über alle Hash-Funktionen auf alle Schlüssel gleichmäßig verteilt.

Mit einer gegebenen Hash-Funktion kollidieren natürlich manche Schlüsselpaare, und andere kollidieren nicht. Aber über viele Instanziierungen unserer Hash-Tabelle hinweg, mit verschiedenen, zufällig ausgewählten Hash-Funktionen, kollidieren alle Schlüsselpaare etwa gleich oft. Genau dies garantieren universelle Hash-Funktionen.

Eine Familie universeller Hash-Funktionen [Slide 26]

$h(k) = (ak + b) \mod p$

mit $p > N$ prim, und a und b werden zufällig gewählt, so dass $0 < a < p$ und $0 \leq b < p$.

Dies ist eine sogenannte *Multiply-Add-Divide* (MAD)-Hash-Funktion.

Anmerkung

Man kann zeigen, dass $c(h(\cdot))$ Hash-Codes mit gleichverteilter Wahrscheinlichkeit über A streut, so dass

$$P(c(h(k)) = n) = \frac{1}{N}$$

$$P(c(h(k_1)) = c(h(k_2))) = \frac{1}{N}$$

über mehrere Abläufe (mit verschiedenen Hash-Funktionen) hinweg.

Beispiel

Hier sehen wir beispielhaft eine Familie universeller Hash-Funktionen. Die Primzahl p wird in Abhängigkeit von N festgesetzt, und die beiden Parameter a und b werden bei jeder Instanziierung einer Hash-Tabelle innerhalb gewisser Grenzen zufällig gewählt.

Man kann beweisen, dass diese Familie von Hash-Funktionen die Definition universeller Hash-Funktionen erfüllt.

Asymptotische Laufzeiten [Slide 27]

Annahmen: gute Hash-Funktion; $n \in \Theta(N)$

Methode	Unsortierte Liste	Hash-Tabelle	
		erwartet	schlechtest
`get`, `put`, `remove`	$O(n)$	$O(1)$	$O(n)$
`entrySet`, `keySet`, `values`	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
`size`, `isEmpty`	$O(1)$	$O(1)$	$O(1)$

Erklärung

Fassen wir hier noch einmal die Laufzeitverhalten der Methoden des abstrakten Datentyps Zuordnungstabelle zusammen. Die Methoden `size()` und `isEmpty()` lassen sich wie immer mit konstanter Laufzeit implementieren. Die Methoden `entrySet()`, `keySet()` und `values()` sind mit einem Schnappschuss-Iterator für die unsortierte Liste und die Hash-Tabelle jeweils $\Theta(n)$, da alle Elemente eingesammelt werden müssen.

Bei der Hash-Tabelle müssen wir hierzu das gesamte Array durchgehen. Hier ist die Laufzeit dieser Methoden daher im Allgemeinen $O(N)$. Man wird jedoch immer die Arraygröße proportional zu n wählen. In diesem Fall sind $O(N)$ und $O(n)$ identisch.

Das Alleinstellungsmerkmal der Hash-Tabelle liegt in der konstanten erwarteten Laufzeit von `get()`, `put()` und `remove()`.

Die schlechtest mögliche Laufzeit dieser Methoden ist jedoch bei Hash-Tabellen nicht besser als bei unsortierten Listen und tritt dann ein, wenn sämtliche Schlüssel auf eine *konstante* Anzahl Felder des Arrays gehasht werden und damit die Hash-Tabelle zu einer Sammlung unsortierter, $O(n)$ langer Listen degradiert. Die Wahrscheinlichkeit, dass ein solcher Fall systematisch eintritt, ist dank universeller Hash-Funktionen extrem gering.

Füllgrade und Rehashing [Slide 28]

Enthält eine Hash-Tabelle der Größe N n Elemente, hat sie einen ***Füllgrad*** *(load factor)* von $\lambda = n/N$.

Für gutes Laufzeitverhalten begrenzt man λ, mindestens $\lambda < 1$.

Experimente und Erwartungswertanalysen legen nahe:

- Mit externer Verkettung begrenze $\lambda < 0.9$.
- Mit offener Adressierung begrenze $\lambda < 0.5$.

Bei Verletzung der Grenze, ***rehash*** in eine Tabelle mindestens *doppelter Größe* (Primzahl) mittels einer *neuen Kompressionsfunktion.*

Anmerkung

- Dies beeinflusst nicht die asymptotischen Laufzeiten.
- Sind die Schlüsselwerte vorab bekannt, kann man ***perfekte*** Hash-Funktionen herleiten.

Erklärung

Diese wunderbaren Eigenschaften der Hash-Tabelle bleiben natürlich nur bestehen, solange ihr ***Füllgrad*** n/N nicht zu groß wird. Andernfalls vergrößern wir das Array auf eine Primzahl, die um ungefähr einen konstanten *Faktor* größer ist als die bisherige Arraygröße, und übertragen die Elemente mittels der entsprechenden, neuen Kompressionsfunktion in das neue Array. Die Hash-Codes können wir beibehalten. Bei universellen Hash-Funktionen wird man allerdings die Gelegenheit nutzen und eine neue Hash-Funktion zufällig generieren.

Diese Prozedur wird ***rehashing*** genannt.

Welchen Füllgrad können wir tolerieren, bevor die Laufzeiten zu degradieren beginnen und wir rehashen sollten?

Experimente und Erwartungswertanalysen legen nahe, dass man bei externer Verkettung den Füllgrad unter 0,9 halten sollte, und bei offener Addressierung unter 0,5.

Wir haben bereits festgestellt, dass ein solches Rehashing die amortisierten asymptotischen Laufzeiten der Methoden nicht beeinflusst.

Zu guter Letzt halten wir noch fest, dass man für eine vorab bekannte, endliche Schlüsselmenge sogenannte ***perfekte*** Hash-Funktionen ermitteln kann, die überhaupt keine Kollisionen verursachen. Damit verbessert sich das Laufzeitverhalten von `get()`, `put()` und `remove()` von $O(n)$ auf $O(1)$ im *schlechtesten* Fall. Da perfekte Hash-Funktionen injektiv sind, kann man die Schlüssel aus ihrem Array-Index erschließen, und man muss lediglich das zugehörige Element speichern.

Anekdote: Hashing und Computersicherheit [Slide 29]

- 2003: Computersicherheitsforscher warnen vor *Denial of Service* durch Hash-Schlüssel, die bei publizierten Hash-Funktionen zu Kollisionen führen.
- Diese Warnung wird weitgehend in den Wind geschlagen.
- 2011: Andere Computersicherheitsforscher demonstrieren einen solchen Angriff auf Web-Server mit URL-Parametern der Form `?key1=val1&key2=val2&key3=val3`, was die Laufzeit von Hash-Tabellen von linearer auf quadratische Laufzeit verschlechtert (in der Anzahl der Parameter).
- 2012: OpenJDK implementiert randomisierte Hash-Funktionen (per Voreinstellung aktiv ab Java 8, 2014).

8 Hash-Tabellen: Implementationen in Java

`AbstractHashMap` [Slide 30]

```
/**
 * An abstract base class supporting Map implementations that use hash
 * tables with MAD compression.
 *
 * The base class provides the following means of support:
 * 1) Support for calculating hash values with MAD compression
 * 2) Support for resizing table when load factor reaches 1/2
 *
 * Subclass is responsible for providing abstract methods:
 *   createTable(), bucketGet(h,k), bucketPut(h,k,v),
 *   bucketRemove(h,k), and entrySet()
 * and for accurately maintaining the protected member, n,
 * to reflect changes within bucketPut and bucketRemove.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public abstract class AbstractHashMap<K,V> extends AbstractMap<K,V> {
  protected int n = 0;              // number of entries in the dictionary
  protected int capacity;           // length of the table
  private int prime;                // prime factor
  private long scale, shift;        // the shift and scaling factors

  /** Creates a hash table with the given capacity and prime factor. */
  public AbstractHashMap(int cap, int p) {
    prime = p;
    capacity = cap;
    Random rand = new Random();
    scale = rand.nextInt(prime-1) + 1;
    shift = rand.nextInt(prime);
    createTable();
  }

  /** Creates a hash table with given capacity and prime factor 109345121. */
  public AbstractHashMap(int cap) { this(cap, 109345121); } // default prime

  /** Creates a hash table with capacity 17 and prime factor 109345121. */
  public AbstractHashMap() { this(17); }              // default capacity

  // public methods
  /**
   * Tests whether the map is empty.
   * @return true if the map is empty, false otherwise
   */
  @Override
  public int size() { return n; }
```

```
/**
 * Returns the value associated with the specified key, or null if no such entry exists.
 * @param key the key whose associated value is to be returned
 * @return the associated value, or null if no such entry exists
 */
@Override
public V get(K key) { return bucketGet(hashValue(key), key); }

/**
 * Removes the entry with the specified key, if present, and returns
 * its associated value. Otherwise does nothing and returns null.
 * @param key the key whose entry is to be removed from the map
 * @return the previous value associated with the removed key, or null if no such entry exists
 */
@Override
public V remove(K key) { return bucketRemove(hashValue(key), key); }

/**
 * Associates the given value with the given key. If an entry with
 * the key was already in the map, this replaced the previous value
 * with the new one and returns the old value. Otherwise, a new
 * entry is added and null is returned.
 * @param key   key with which the specified value is to be associated
 * @param value value to be associated with the specified key
 * @return the previous value associated with the key (or null, if no such entry)
 */
@Override
public V put(K key, V value) {
  V answer = bucketPut(hashValue(key), key, value);
  if (n > capacity / 2)          // keep load factor <= 0.5
    resize(2 * capacity - 1);    // (or find a nearby prime)
  return answer;
}

// private utilities
/** Hash function applying MAD method to default hash code. */
private int hashValue(K key) {
  return (int) ((Math.abs(key.hashCode()*scale + shift) % prime) % capacity);
}

/** Updates the size of the hash table and rehashes all entries. */
private void resize(int newCap) {
  ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
  for (Entry<K,V> e : entrySet())
    buffer.add(e);
  capacity = newCap;
  createTable();                 // based on updated capacity
  n = 0;                         // will be recomputed while reinserting entries
  for (Entry<K,V> e : buffer)
    put(e.getKey(), e.getValue());
}
```

```
  // protected abstract methods to be implemented by subclasses
  /** Creates an empty table having length equal to current capacity. */
  protected abstract void createTable();

  /**
   * Returns value associated with key k in bucket with hash value h.
   * If no such entry exists, returns null.
   * @param h the hash value of the relevant bucket
   * @param k the key of interest
   * @return  associate value (or null, if no such entry)
   */
  protected abstract V bucketGet(int h, K k);

  /**
   * Associates key k with value v in bucket with hash value h, returning
   * the previously associated value, if any.
   * @param h the hash value of the relevant bucket
   * @param k the key of interest
   * @param v the value to be associated
   * @return  previous value associated with k (or null, if no such entry)
   */
  protected abstract V bucketPut(int h, K k, V v);

  /**
   * Removes entry having key k from bucket with hash value h, returning
   * the previously associated value, if found.
   * @param h the hash value of the relevant bucket
   * @param k the key of interest
   * @return  previous value associated with k (or null, if no such entry)
   */
  protected abstract V bucketRemove(int h, K k);
}
```

ChainHashMap: externe Verkettung [Slide 31]

```
public class ChainHashMap<K,V> extends AbstractHashMap<K,V> {
  // a fixed capacity array of UnsortedTableMap that serve as buckets
  private UnsortedTableMap<K,V>[] table; // initialized within createTable

  // provide same constructors as base class
  /** Creates a hash table with capacity 11 and prime factor 109345121. */
  public ChainHashMap() { super(); }

  /** Creates a hash table with given capacity and prime factor 109345121. */
  public ChainHashMap(int cap) { super(cap); }

  /** Creates a hash table with the given capacity and prime factor. */
  public ChainHashMap(int cap, int p) { super(cap, p); }

  /** Creates an empty table having length equal to current capacity. */
  @Override
  @SuppressWarnings({"unchecked"})
  protected void createTable() {
    table = (UnsortedTableMap<K,V>[]) new UnsortedTableMap[capacity];
  }

  /**
   * Returns value associated with key k in bucket with hash value h.
   * If no such entry exists, returns null.
   * @param h the hash value of the relevant bucket
   * @param k the key of interest
   * @return  associate value (or null, if no such entry)
   */
  @Override
  protected V bucketGet(int h, K k) {
    UnsortedTableMap<K,V> bucket = table[h];
    if (bucket == null) return null;
    return bucket.get(k);
  }

  /**
   * Associates key k with value v in bucket with hash value h, returning
   * the previously associated value, if any.
   * @param h the hash value of the relevant bucket
   * @param k the key of interest
   * @param v the value to be associated
   * @return  previous value associated with k (or null, if no such entry)
   */
  @Override
  protected V bucketPut(int h, K k, V v) {
    UnsortedTableMap<K,V> bucket = table[h];
    if (bucket == null)
      bucket = table[h] = new UnsortedTableMap<>();
    int oldSize = bucket.size();
    V answer = bucket.put(k,v);
```

```
    n += (bucket.size() - oldSize); // size may have increased
    return answer;
  }

  /**
   * Removes entry having key k from bucket with hash value h, returning
   * the previously associated value, if found.
   * @param h the hash value of the relevant bucket
   * @param k the key of interest
   * @return  previous value associated with k (or null, if no such entry)
   */
  @Override
  protected V bucketRemove(int h, K k) {
    UnsortedTableMap<K,V> bucket = table[h];
    if (bucket == null) return null;
    int oldSize = bucket.size();
    V answer = bucket.remove(k);
    n -= (oldSize - bucket.size()); // size may have decreased
    return answer;
  }

  /**
   * Returns an iterable collection of all key—value entries of the map.
   *
   * @return iterable collection of the map's entries
   */
  @Override
  public Iterable<Entry<K,V>> entrySet() {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>();
    for (int h=0; h < capacity; h++)
      if (table[h] != null)
        for (Entry<K,V> entry : table[h].entrySet())
          buffer.add(entry);
    return buffer;
  }
}
```

ProbeHashMap: lineare Sondierung [Slide 32]

```
public class ProbeHashMap<K,V> extends AbstractHashMap<K,V> {
  private MapEntry<K,V>[] table;   // a fixed array of entries (all initially null)
  private MapEntry<K,V> DEFUNCT = new MapEntry<>(null, null); //sentinel

  // provide same constructors as base class
  /** Creates a hash table with capacity 17 and prime factor 109345121. */
  public ProbeHashMap() { super(); }

  /** Creates a hash table with given capacity and prime factor 109345121. */
  public ProbeHashMap(int cap) { super(cap); }

  /** Creates a hash table with the given capacity and prime factor. */
  public ProbeHashMap(int cap, int p) { super(cap, p); }

  /** Creates an empty table having length equal to current capacity. */
  @Override
  @SuppressWarnings({"unchecked"})
  protected void createTable() {
    table = (MapEntry<K,V>[]) new MapEntry[capacity]; // safe cast
  }

  /** Returns true if location is either empty or the "defunct" sentinel. */
  private boolean isAvailable(int j) {
    return (table[j] == null || table[j] == DEFUNCT);
  }

  /**
   * Searches for an entry with key equal to k (which is known to have
   * hash value h), returning the index at which it was found, or
   * returning -(a+1) where a is the index of the first empty or
   * available slot that can be used to store a new such entry.
   *
   * @param h the precalculated hash value of the given key
   * @param k the key
   * @return index of found entry or if not found, value -(a+1)
   *         where a is the index of the first available slot
   */
  private int findSlot(int h, K k) {
    int avail = -1;                          // no slot available (thus far)
    int j = h;                               // index while scanning table
    do {
      if (isAvailable(j)) {                  // may be either empty or defunct
        if (avail == -1) avail = j;          // this is the first available slot!
        if (table[j] == null) break;         // if empty, search fails immediately
      } else if (table[j].getKey().equals(k))
        return j;                            // successful match
      j = (j+1) % capacity;                  // keep looking (cyclically)
    } while (j != h);                        // stop if we return to the start
    return -(avail + 1);                     // search has failed
  }
```

```
/**
 * Returns value associated with key k in bucket with hash value h.
 * If no such entry exists, returns null.
 * @param h the hash value of the relevant bucket
 * @param k the key of interest
 * @return  associate value (or null, if no such entry)
 */
@Override
protected V bucketGet(int h, K k) {
  int j = findSlot(h, k);
  if (j < 0) return null;              // no match found
  return table[j].getValue();
}

/**
 * Associates key k with value v in bucket with hash value h, returning
 * the previously associated value, if any.
 * @param h the hash value of the relevant bucket
 * @param k the key of interest
 * @param v the value to be associated
 * @return  previous value associated with k (or null, if no such entry)
 */
@Override
protected V bucketPut(int h, K k, V v) {
  int j = findSlot(h, k);
  if (j >= 0)                          // this key has an existing entry
    return table[j].setValue(v);
  table[-(j+1)] = new MapEntry<>(k, v); // convert to proper index
  n++;
  return null;
}

/**
 * Removes entry having key k from bucket with hash value h, returning
 * the previously associated value, if found.
 * @param h the hash value of the relevant bucket
 * @param k the key of interest
 * @return  previous value associated with k (or null, if no such entry)
 */
@Override
protected V bucketRemove(int h, K k) {
  int j = findSlot(h, k);
  if (j < 0) return null;              // nothing to remove
  V answer = table[j].getValue();
  table[j] = DEFUNCT;                  // mark this slot as deactivated
  n--;
  return answer;
}

/**
 * Returns an iterable collection of all key—value entries of the map.
```

```
   *
   * @return iterable collection of the map's entries
   */
  @Override
  public Iterable<Entry<K,V>> entrySet() {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>();
    for (int h=0; h < capacity; h++)
      if (!isAvailable(h)) buffer.add(table[h]);
    return buffer;
  }
}
```

9 Verwandte ADTs

ADT: Sortierte Zuordnungstabelle [Slide 33]

Zuordnungstabelle, plus:

```
firstEntry()    // Returns the entry with smallest key (or null)
lastEntry()     // Returns the entry with largest key (or null)

floorEntry(k)   // Returns the entry with the greatest key ≤ k (or null)
ceilingEntry(k) // Returns the entry with the least key ≥ k (or null)

lowerEntry(k)   // Returns the entry with the greatest key < k (or null)
higherEntry(k)  // Returns the entry with the least key > k (or null)

subMap(k1, k2)  // Returns an iteration of all entries
                // with k1 ≤ key < k2
```

Laufzeiten einer Implementierung als sortiertes Array?

Anwendungsbeispiele?

Kontaktliste

Quiz [Slide 34]

Lassen sich sortierte Zuordnungstabellen auf Basis einer Hash-Tabelle implementieren?

A: Ja, auf recht einfache Weise.

B: Ja, aber mit substanziellem Aufwand.

C: Nein, man benötigt eine komplett andere Datenstruktur.

D: weiß nicht

ADT: Menge *(Set)* [Slide 35]

```
add(e)       // Adds the element e to S (if not already present)
remove(e)    // Removes the element e from S (if present)
contains(e)  // Returns whether e ∈ S
iterator()   // Returns an iterator of the elements of S

addAll(T)    // Replaces S by S ∪ T
retainAll(T) // Replaces S by S ∩ T
removeAll(T) // Replaces S by S − T
```

```
public void addAll(Set<E> other) {
  for (E element : other) // rely on iterator() method of other
    add(element);         // duplicates will be dropped by add()
}
```

ADT: Sortierte Menge [Slide 36]

Menge, plus:

```
first()        // Returns smallest element (or null)
last()         // Returns largest element (or null)

floor(e)       // Returns the greatest element ≤ e (or null)
ceiling(e)     // Returns the least element ≥ e (or null)

lower(e)       // Returns the greatest element < e (or null)
higher(e)      // Returns the least element > e (or null)

subSet(e1, e2) // Returns an iteration of all elements
               // with e1 ≤ element < e2

pollFirst()    // Returns and removes the smallest element
pollLast()     // Returns and removes the largest element
```

Mengen implementieren [Slide 37]

Wie Zuordnungstabellen, aber ohne Unterscheidung zwischen Schlüssel und Wert.

MultiSet, MultiMap [Slide 38]

Kernfrage: Gelten ***gleiche*** Elemente als *identisch* oder lediglich als *äquivalent*?

- *äquivalent*: Instanzen werden separat abgespeichert.

  Z.B. in einem sekundären Container unter einem Repräsentanten (Element bzw. Schlüssel), oder als separate Instanzen.

- *identisch*: Jeweils eine Instanz wird abgespeichert, mit Zähler.

10 Zusammenfassung

Zusammenfassung [Slide 39]

ADT:

- Zuordnungstabelle
- Sortierte Zuordnungstabelle
- Menge; Sortierte Menge

Algorithmen:

- Hashing
  - Hash-Funktion = Kompressionsfunktion(Hash-Code(Schlüssel))
  - Kollisionsbehandlung: externe Verkettung, offene Adressierung
  - Füllgrade und Rehashing

Literatur [Slide 40]

Goodrich, Michael, Roberto Tamassia und Michael Goldwasser (Aug. 2014). *Data Structures and Algorithms in Java*. Wiley.