TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

Abstrakte Datentypen: Maps

Abstrakte Datentypen: Maps

Indexstrukturen (engl. *maps* oder *dictionary*) erlauben eine effiziente Verwaltung von Daten (hier: Objekten). Ein Datensatz besteht aus einem eindeutigen Schlüssel und dem ihm zugeordneten Wert.
Die Verwaltung von Datensätzen basiert auf den folgenden drei Grundoperationen:

- Einfügen eines Datensatzes in eine Menge von Datensätzen
- Suchen eines Datensatzes mit Schlüssel k
- Löschen eines Datensatzes mit Schlüssel k

Implementierungen für Maps

In vereinfachter Anlehnung an `java.util.Map` legen wir für die hier diskutierten Implementierungen folgende Schnittstelle zugrunde:

```
interface Map {
    Object get(int key);
    void   put(int key, Object value);
    void   remove(int key);
}
```

Um die Implementierung zu vereinfachen, betrachten wir zunächst eine nicht-parametrisierte Variante mit `int`-Schlüsseln und `Object`-Daten.

Beispiel: Abbildung Zahl auf Monatsname

```
Map monate = new ... // Implementierung folgt spaeter
monate.put(1, "Januar");
monate.put(2, "Februar");
...
monate.put(12, "Dezember");
System.out.println("Monat: " + monate.get(4));
// Monat: April
```

Implementierungen für Maps

Ziel ist es, *Datenstrukturen* für die Implementierung auszuwählen, bei denen der Aufwand für obige Operationen gering ist.

Hier behandeln wir drei klassische Implementierungen, bei denen Suchen, Einfügen und Entfernen effizient durchgeführt werden kann:

- A. Binäre Suche in Arrays
- B. Balancierte Suchbäume
- C. Hashing

Dabei betracheten wir jeweils

- die Datenstruktur
- die drei grundlegenden Operationen
- eine einfache Komplexitätsabschätzung

A. Binäre Suche in Arrays

Bei dieser Implementierung verwenden wir ein Array von Datensätzen, welches nach den Schlüsselwerten sortiert ist. In diesem kann ein Datensatz mit Hilfe von binärer Suche effizient gefunden werden.

Ein Datensatz wird durch folgende Klasse repräsentiert:

```
class DataSet {
  int key;
  Object data;
  DataSet (int k, Object d) {
    key = k;
    data = d;
  }
}
```

Datenstruktur I

Eine Map wird repräsentiert durch ein Objekt mit:

- einer Referenz auf das Array mit den Datensätzen
- der Größenangabe des Arrays (`capacity`)
- der Anzahl der gespeicherten Datensätze (`size`)

Die Operationen gewährleisten folgende Invariante:

- Die Datensätze sind aufsteigend nach Schlüssel sortiert.
- Die Schlüssel sind eindeutig.

Datenstruktur II

```
public class ArrayMap implements Map {
  private DataSet[] elems;
  private int   capacity;
  private int   size;
  public ArrayMap() {
    elems    = new DataSet[8];
    capacity = 8;
    size     = 0;
  }
  private int searchIndex( int key )          { ... }
  public Object get( int key )                { ... }
  public void remove( int key )               { ... }
  public void put( int key, Object value ) {...}
}
```

Datenstruktur III

Zum Einfügen, Suchen und Löschen benötigt man den Index, an dem die Operation ausgeführt werden soll:

```
/* ensures
     0 <= \result <= size
     && fuer alle int i in [0, \result - 1] gilt:
          elems[i].key < key
     && fuer alle int i in [\result, size-1] gilt:
          elems[i].key >= key
 */
private int searchIndex(int key) {
     ...
}
```

Insbesondere gilt damit, dass `searchIndex(k)` den Index `i` in `elems` liefert, sodass `elems[i].key == k`, falls so ein Datensatz existiert.

Heraussuchen

```
public Object get ( int key ) {
  int ix = searchIndex( key );
  if( ix == size || elems[ix].key != key ){
    return null;
  } else {
    return elems[ix].data;
  }
}
```

Löschen

```
public void remove( int key ) {
  int ix = searchIndex( key );
  if( ix != size && elems[ix].key == key ){
    /* Datensatz loeschen */
    for( int i = ix + 1; i < size; i++ ) {
      elems[i-1] = elems[i];
    }
    size--;
  }
}
```

Einfügen I

```
public void put( int key, Object value ) {
   int ix = searchIndex( key );
   if( ix == size || elems[ix].key > key ) {
      /* neuen Datensatz eintragen */
      size++;
      if( size > capacity ) { // neues Feld anlegen
         DataSet[] newElems = new DataSet[2*capacity];
         for( int i = 0; i < ix; i++ ) {
            newElems[i] = elems[i];
         }
         for( int i = ix + 1; i < size; i++ ) {
            newElems[i] = elems[i-1];
         }
         elems = newElems;
         capacity = 2*capacity;
```

Einfügen II

```
      } else { // Elemente im Feld verschieben
        for( int i = size-1; i>=ix+1; i-- ) {
          elems[i] = elems[i-1];
        }
      }
      elems[ix] = new DataSet( key, value );
    } else { // elems[ix].key == key
      elems[ix].data = value;
    }
}
```

Bemerkung:

Allen Operationen liegt die Suchoperation `searchIndex(key)` zugrunde. Daher ist eine schnelle Suche wichtig. Deshalb konzentrieren sich die algorithmischen Untersuchungen auf diese Operation.

Wiederholung: Binäre Suche

```
private int searchIndex(int key) {
    int ug = 0;
    int og = size-1;

    // fuer alle int i < ug gilt elems[i].key < key
    // fuer alle int i > og gilt elems[i].key > key
    while (ug <= og) {
        int mid = ug + (og-ug)/2;
        if (key < elems[mid].key) {
            og = mid - 1;
        } else if (key > elems[mid].key) {
            ug = mid + 1;
        } else {
            return mid;
        }
    }
    // ug == og + 1
    // fuer alle int i < ug gilt elems[i].key < key
    // fuer alle int i >= ug gilt elems[i].key > key

    return ug;
}
```

Diskussion

Binäres Suchen verursacht logarithmischen Aufwand: $O(\log N)$. Damit auch das Herausholen eines Eintrags aus der ArrayMap.

Einfügen und Löschen benötigen in der gezeigten Variante linearen Aufwand: $O(N)$.

Vorteile:

- einfach und speichersparend zu realisieren
- schnelles Heraussuchen von Einträgen

Nachteile:

- Einfügen und Löschen sind vergleichsweise langsam.

B. Balancierte Suchbäume

In Abschnitt 12 haben wir natürliche binäre Suchbäume (sortierte markierte Binärbäume) betrachtet. Sofern binäre Suchbäume hinreichend gut ausgeglichen (*balanciert*) sind, ist der Aufwand aller drei Grundoperationen logarithmisch.

Ziel ist es, bei den modifizierenden Operationen (Einfügen und Entfernen) den Baum wenn nötig wieder auszubalancieren.

Durch Anforderungen bzgl. einer Verteilung der Blätter und Höhen in Unterbäumen kann man ein Degenerieren verhindern.

- Vorteil: geringer Aufwand für Grundoperationen kann zugesichert werden
- Nachteil: Strukturinvariante muss erhalten werden

Beispiel: AVL-Baum I

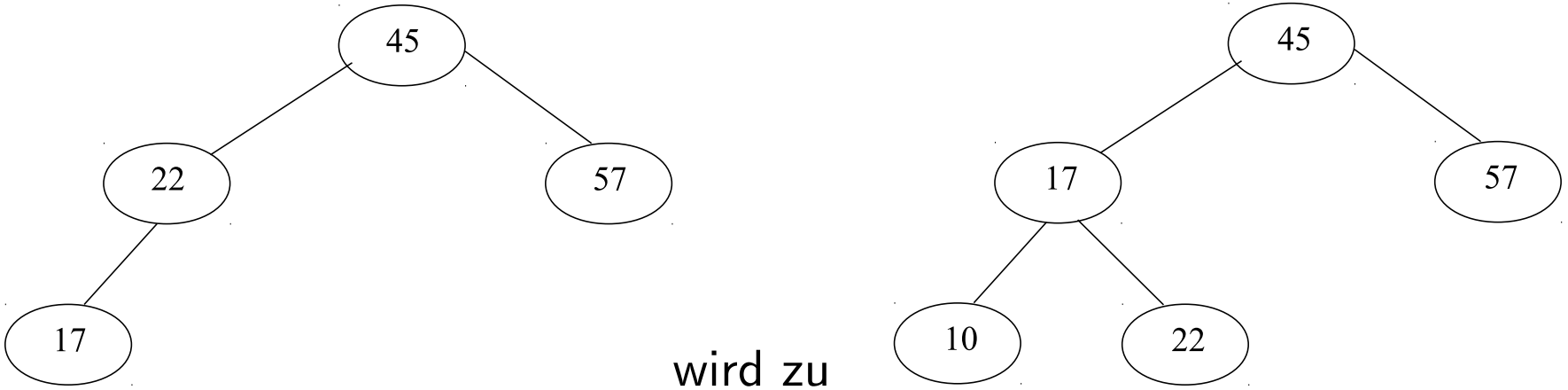
Adelson-Velskij und Landis schlugen folgende Balancierungseigenschaft vor:

Ein binärer Suchbaum heißt höhenbalanciert, wenn für jeden Knoten K gilt:

> *Die Höhe des linken Unterbaums von K unterscheidet sich von der Höhe des rechten Unterbaums höchstens um eins.*

Beispiel: AVL-Baum II

Das Einfügen von Schlüssel 10 erfordert ein Vertauschen (Rotieren) der Knoten im linken Teilbaum:



Diskussion: Balancierte Suchbäume

Bei Balancierten Suchbäumen verursachen die Such-, Einfüge- und Löschoperation (inkl. Rebalancierungsoperationen) jeweils logarithmischen Aufwand: $O(\log N)$

Die Literatur listet eine Reihe verschiedener Balancierungsstrategien (RedBlack-Trees, AVL-Trees, B-Trees, Splay-Trees, etc.).

C. Hashing/Streuspeicherung

Anstatt durch schrittweises Vergleichen von Schlüsseln auf einen Datensatz zuzugreifen, versucht man bei Hash- oder Streuspeicherverfahren aus dem Schlüssel die Positionsinformation des Datensatzes (z.B. den Arrayindex) zu *berechnen*.

Für viele praktisch relevante Szenarien erreicht man dadurch Datenzugriff mit *konstantem* Aufwand.

Begriffsklärung: Hashfunktion, -tabelle I

Seien

- S die Menge der möglichen Schlüsselwerte (*Schlüsselraum*) und
- A die Menge von Adressen in einer Hashtabelle (im Folgenden ist A immer die Indexmenge $0 \ldots m-1$ eines Feldes).

Eine **Hashfunktion** $h : S \to A$ ordnet jedem Schlüssel eine Adresse in der Hashtabelle zu.

Als **Hashtabelle (HT)** der Größe m bezeichnen wir einen Speicherbereich, auf den über die Adressen aus A mit konstantem Aufwand (also unabhängig von m) zugegriffen werden kann.

Begriffsklärung: Hashfunktion, -tabelle II

Enthält S weniger Elemente als A, kann h injektiv sein:

Für alle s, t in $S : s \neq t \Rightarrow h(s) \neq h(t)$

d.h. die Hashfunktion ordnet jedem Schlüssel eine eineindeutige Adresse zu. Dann ist perfektes Hashing möglich.

Andernfalls können Kollisionen auftreten.

Begriffsklärung: Kollision, Synonym

Zwei Schlüssel s, t **kollidieren** bezüglich einer Hashfunktion h, wenn $h(s) = h(t)$.

Die Schlüssel s und t nennt man dann **Synonyme**.

Die Menge der Synonyme bezüglich einer Adresse a aus A heißt die **Kollisionsklasse** von a.

Ist schon ein Datensatz mit Schlüssel s in der Hashtabelle gespeichert, nennt man einen Datensatz mit einem Synonym von s einen **Überläufer**.

Anforderungen an Hashfunktionen

Eine Hashfunktion soll

- sich einfach und effizient berechnen lassen
- zu einer möglichst gleichmäßigen Belegung der Hashtabelle führen
- möglichst wenige Kollisionen verursachen

Klassifikation von Hashverfahren

Hashverfahren unterscheiden sich

- durch die Hashfunktion
- durch die Kollisionsauflösung:
  - **Verkettung**: Überläufer werden in einer Liste an der Position der Hashtabelle eingefügt
  - **offen**: Überläufer werden an noch offenen Positionen der Hashtabelle gespeichert
- durch die Wahl der Größe der Hashtabelle:
  - **statisch**: Die Größe wird bei der Erzeugung festgelegt und bleibt unverändert.
  - **dynamisch**: Die Größe kann angepasst werden.

Wir betrachten im Folgenden eine Realisierung einer statischen Hashtabelle mit Kollisionsauflösung durch Verkettung.

Hashfunktion

Entscheidend ist, dass die Hashfunktion die Schlüssel gut streut.

Verbreitetes Verfahren:

- Wähle eine Primzahl als Hashtabellen-Größe.
- Wähle den ganzzahligen Divisionsrest als Hashwert:

```
private  int hash( int key ) {
   return Math.abs(key % hashtable.length);
 }
```

Datenstruktur I

Ein Eintrag in der Hashtabelle wird durch ein Objekt der folgenden Klasse repräsentiert:

```
class HashEntry {
  int key;
  Object value;
  HashEntry next;
  HashEntry(int k, Object v) {
    this.key = k;
    this.value = v;
  }
}
```

Datenstruktur II

Wir realisieren eine Hashtabelle als Implementierung der Schnittstelle Map:

```
class HashMap implements Map {
  private HashEntry[] table;

  public HashMap( int tabsize ) {
    /* tabsize sollte eine Primzahl sein */
    this.table = new HashEntry[tabsize];
  }
  private  int hash( int key ) { ... }
  public Object get( int key ) { ... }
  public void put (int key,Object v ){ ... }
  public void remove( int key ) { ... }
}
```

Datenstruktur III

Sei s ein Schlüssel, $h(s)$ sein Hashwert. Es werden nur Objekte eingetragen (`value`-Parameter von `put` ist immer ungleich `null`).
Der Datenstruktur HashMap liegen folgende Invarianten zugrunde:

- Die Hashtabelle enthält den Datensatz zu s, wenn
  - → `table[ h(s) ]  != null` und
  - → für ein Elemente `entry` der verlinkten Liste startend bei `table[ h(s) ]` gilt: `entry.key == s`

  Die Daten liefert dann `entry.value` .

- Das heißt: Alle Elemente der Kollisionsklasse zu $h(s)$ befinden sich in der Liste an Position $h(s)$ der Hashtabelle.

Beim Setzen von Einträgen in der Hashtabelle muss man sich entscheiden, ob man bestehende Einträge mit gleichem Schlüssel ersetzt (höhere Laufzeit), oder den Eintrag zunächst hinzufügt und alte Einträge erst bei Gelegenheit entfernt (höherer Speicherbedarf).

Einfügen

```
public void put ( int key, Object value ) {
  if(value != null) {
    int hix = hash(key);
    HashEntry currentEntry = table[hix];
    HashEntry newEntry = new HashEntry(key, value);
    newEntry.next = currentEntry;
    table[hix] = newEntry;
  }
}
```

Suchen

```
public Object get( int key ) {
  int hix = hash(key);
  HashEntry entry = table[hix];

  while(entry != null) {
    if (entry.key == key) {
      return entry.value;
    }
    entry = entry.next;
  }
  return null;
}
```

Löschen

```
public void remove( int key ) {
    int hix = hash(key);
    HashEntry entry = table[hix];

    // entfernen alle HashEntries mit dem gegebenen Schluessel
    HashEntry dummy = new HashEntry(0, null);
    dummy.next = entry;
    HashEntry current = dummy;
    while(current.next != null) {
      if(current.next.key == key) { // entferne den Eintrag
        current.next = current.next.next;
      } else { // gehe zum naechsten Eintrag
        current = current.next;
      }
    }
    table[hix] = dummy.next;
  }
```

Diskussion

Die Komplexität der Operationen einer Hashtabelle ist abhängig von

- der Hashfunktion und dem Füllungsgrad der Tabelle
- dem Verfahren zur Kollisionsauflösung

Bei guter Hashfunktion und kleinem Füllungsgrad kommt man im Mittel mit konstantem Aufwand $O(1)$ aus.

Bemerkung

Die gezeigte Implementierung ist nicht optimal. Im schlechtesten Fall haben die Operationen einen Aufwand in $O(n)$ mit n die Anzahl der Einträge in der Hashtabelle.

Mögliche Optimierungen:

- Sortierung der `HashEntry`-Liste
- Implementierung der Kollisionsauflösung mit Hilfe eines Suchbaums

Generische Maps

Generische Maps

Im Allgemeinen möchte man sich nicht auf Schlüssel des Typs `int` und Werte des Typs `Object` beschränken. Man möchte eine generische Variante des Interfaces `Map` verwenden.

Das Java Collection Framework beinhaltet eine solche generische Variante des `Map` Interfaces: `java.util.Map<K, V>`.

Maps im Java Collections Framework

Die wichtigsten Methoden:

```
interface Map<K,V> {
  // Liefert das Objekt unter Schluessel key;
  // falls nicht vorhanden, wird null zurueckgegeben
  V get(Object key);

  // Fuegt den Wert value unter Schluessel key ein
  // liefert das Objekt, der zuvor unter key eingetragen war;
  // falls nicht vorhanden, wird null zurueckgegeben
  V put(K key, V value);

  // Entfernt den Eintrag unter Schluessel key;
  // Liefert das bisher eingetragene Objekt
  // falls nicht vorhanden, wird null zurueckgegeben
  V remove(Object key);

  // Liefert ein Set mit allen eingetragenen Schluesseln
  Set<K>  keySet();

  // Liefert ein Set mit allen Eintraegen
  Set<Map.Entry<K,V>> entrySet();
  ...
}
```

Bemerkungen

- Das `Map`-Interface ist parametrisiert über `K` (Typ der Schlüssel) und `V` (Typ der Daten).
- Wichtige implementierende Klassen: `TreeMap`, `HashMap`
- Die Einträge der `Map` sind vom Typ `Map.Entry<K,V>`.
  - Zugriff auf Schlüssel eines Eintrags: `K getKey()`
  - Zugriff auf Daten eines Eintrags: `V getValue()`

Hinweis: Beim Typ `Map.Entry<K, V>` handelt es sich um eine innere Klasse. Diese besprechen wir im Anschluss.

Hashen von Objekten

Bis jetzt haben wir als Schlüssel nur ganzzahlige Werte betrachtet. Bei der generischen Implementierung sind Schlüssel von einem Objekttyp.

Ein als Objekttyp vorliegender Schlüssel muss in einen numerischen Wert umgewandelt werden (Hashadresse). In Java implementiert und verwendet man dazu die Methode `int hashCode()` der Klasse `Object`.

Soll eine Klasse als Typ eines Schlüssels in einer HashMap verwendet werden, so muss die `equals()` konsistent mit der `hashCode()` Methode sein.

Hashen von Objekten (Code)

```
class HashMap<K, V> implements Map<K, V> { ...
  private int hash( K key ) {
    return Math.abs(key.hashCode() % table.length);
  }
  public V get( K key ) {
    int hix = hash(key);
    HashEntry<K, V> entry = table[hix];

    while(entry != null) {
      if (entry.key.equals(key)) {
        return entry.value;
      }
      entry = entry.next;
    }
    return null;
  }
}
```

Bemerkungen

- Die Implementierung einer `hashCode()` Methode, die die Ergebnisse optimal streut, ist nicht einfach.
- Die Implementierungen von `equals()` und `hashCode()` müssen zueinander passen, gleiche Objekte müssen den gleichen Hashcode haben.

Einschub: Geschachtelte Klassen

Begriffsklärung: Geschachtelte Klassen I

Eine Klasse heißt in Java **geschachtelt** (engl. *nested*), wenn sie innerhalb der Deklaration einer anderen Klasse deklariert ist:

- als Komponente (ähnlich einem Attribut),
- als **lokale** Klassen (ähnlich einer lokalen Variablen)
- als **anonyme** Klassen bei der Objekterzeugung.

Ist eine Klasse nicht geschachtelt, nennen wir sie **global** (engl. *top-level*).

Java unterscheidet statische und innere Klassen.

Begriffsklärung: Geschachtelte Klassen II

Geschachtelte Klassen heißen **statisch**, wenn sie mit dem Modifikator `static` deklariert sind.

Statische Klasse haben die gleiche Bedeutung wie globale Klassen. Allerdings ergeben sich andere Sichtbarkeits- und Zugriffsbereiche.

Insbesondere:

- Geschachtelte Klassen können als privat deklariert werden, so dass sie außerhalb der umfassenden Klasse nicht verwendet werden können.
- Geschachtelte Klassen können auf private Attribute und Methoden der umfassenden Klasse zugreifen.

Beispiel: Statische Klassen

```
public class LinkedList {
    private Node entries = null;
    private int size = 0;

    private static class Node {
        int elem;
        Node next;
    }
    int getFirst() { ... }
    ...
}
```

Der Typ `Node` ist nur innerhalb von `LinkedList` sichtbar und zugreifbar.

Bemerkung

- Geschachtelte Klassen können auch erben, Interfaces implementieren und generisch sein.
- **Achtung:** eine innere Klasse übernimmt **nicht** die Superklassen und Interfaces der umschließenden Klasse!
- Die Zugriffs- und Sichtbarkeitsregeln im Zusammenhang mit geschachtelten Klassen sind recht komplex.
  - Äußere Klassen können auf private Elemente in inneren Klassen zugreifen.
  - Innere Klassen können auf private Elemente in äußeren Klassen zugreifen.
  - Innere Klassen können auf private Elemente anderer innerer Klassen mit gleicher äußerer Klasse zugreifen.
- Der Compiler erzeugt getrennte .class-Dateien (z.B. `LinkedList$Node`).
- Zugreifbare geschachtelte Klassen können über zusammengesetzte Namen außerhalb der umfassenden Klasse angesprochen werden (z.B. `LinkedList.Node`).

Zusammenfassung: Die wichtigsten Collections

Interface	Hashtable	Resizable Array	Balanced Tree	LinkedList	Hashtable + LinkedList
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Quelle: https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html