Datenstrukturen

Dr Steffen Jost

Institut für Informatik der LMU München

Datenstrukturen

Datenstrukturen legen fest, wie Daten im Speicher angeordnet werden und nach welchem Muster darauf zugegriffen wird.

Modulares Design von Datenstrukturen:

Interface	legt Funktionalität fest
Implementierung	legt Effizienz fest

Austausch der Implementierung darf Performanz beeinflussen, nicht jedoch semantischen Korrektheit!

Datenstruktur Beispiele

Beispiele für beliebte Datenstrukturen in Java

Struktur	Interface	Implementierung
Menge	`Set<E>`	`HashSet`, `TreeSet`, `EnumSet`, ...
Abbildung	`Map<K,V>`	`HashMap`, `TreeMap`, `EnumMap`, ...
Liste	`List<E>`	`ArrayList`, `LinkedList`, `Stack`, ...

Implementierungen können Interface spezialisieren, z.B. durch Forderung von Zusatzeigenschaften an Typ der verwalteten Daten: `EnumSet<E extends Enum<E>>`

Endliche Mengen

Endliche Menge von verschiedenen Elementen gleichen Typs

- Gleichheit muss verfügbar sein
- Elemente maximal einmal enthalten — sonst: Bag / MultiSet
- Reihenfolge unerheblich — sonst: `LinkedHashSet`

Funktionalität

- Test, ob Element in Menge enthalten ist — Primäre Op.
- Hinzufügen / Entfernen eines Elementes
- Bildung von Vereinigungs- / Schnittmenge
- Iteration über Elemente der Menge

Interface `Set<E>`


```
interface Set<E> extends Collection<E>
  boolean contains(Object o);
  boolean add(E e);
  boolean addAll(Collection<? extends E> c);
  boolean remove(Object o);
  boolean removeAll(Collection<?> c);
  boolean retainAll(Collection<?> c);
  Iterator<E> iterator();
  ...
```

- Methoden Signaturen wie im Interface `Collection`, teilweise jedoch stärkere Bedingungen
- Objekte sollten möglichst immutable sein
- Vergleiche benutzen Methode `equals`
- Implementierung mittels Hashing oder Suchbäumen

equals & hashCode

Jedes Java Objekt bietet die Methoden

```
boolean equals(Object obj);
int     hashCode();
```

`equals` vergleicht zwei Objekte,
`hashCode` berechnet möglichst eindeutigen "Fingerabdruck"

Spezifikation schreibt vor:

"Gleiche" Objekte müssen gleichen Hash Code liefern

Interface `Set` spezifiziert `equals`, Implementierungen nutzen oft `hashCode` und verlassen sich auf diesen Kontrakt

- immer beide Methoden überschreiben — Eclipse generiert
- Veränderung von Objekten im Set dürfen Ergebnis dieser Methoden nicht beeinflussen

equals

`equals` sollte Äquivalenzrelation sinnvoll kodieren,
d.h. falls `x`, `y`, `z` nicht null gilt:

1. Reflexiv: `x.equals(x)` liefert `True`
2. Symmetrisch: `x.equals(y)` gleich `y.equals(x)`
3. Transitiv: `x.equals(y)` und `y.equals(z)` impliziert `x.equals(z)`
4. Deterministisch: Wiederholter Test liefert gleiches Ergebnis bei unveränderten Objekten
5. Non-Null: `x.equals(null)` liefert `False`, keine Exception

Von `Object` geerbte Methode vergleicht Referenzen; wie `x==y`

Demo: `equals`

Code Demonstration:

- Erstellen der Klasse `Position` `Paar<int,Integer>`
- Autogenerierung von `equals` und `hashCode`
- Erstellen einer Unterklasse `ColorPosition`
- Probleme mit Vererbung diskutieren, Verletzung von Symmetrie & Transitivität

`equals` & Vererbung

Wenn eine Unterklasse `equals` erneut überschreibt, dann kann dieses nicht mehr symmetrisch und transitiv sein

Lösungen:

- Erbe überschreibt `equals` nicht — ignoriert neue Attribute
- `equals` in Oberklasse identisch zu Referenzvergleich `==`
- Oberklasse ist abstrakt, d.h. keine Objekte möglich
- Komposition anstatt Vererbung benutzen.
  Ermöglicht jeweils differenzierte `equals` Methoden,
  welche dann bewusst ausgewählt werden.

Wenn Klasse `equals` überschreibt,
dann sollte diese `abstract` sein oder Vererbung verbieten
(d.h. Klasse `final` oder alle Konstruktoren `private`)

hashCode

Wenn `equals` überschrieben wird, dann auch immer `hashCode`!

Es *muss* gelten:

`x.equals(y)` impliziert `x.hashCode() == y.hashCode()`

Es *sollte* gelten (aber muss nicht):

`x.hashCode() == y.hashCode()` impliziert `x.equals(y)`

Aus erster Forderung folgt, dass Hashwerte bei Änderungen am Objekt neu berechnet werden müssen.
Unveränderliche Objekte können Hashwerte cachen.

Konstante Hashwerte erfüllt Mindestanforderung, kann sehr viel Performanz kosten: `public int hashCode() { return 0; };`

Demo: `hashCode`

Code Demonstration:

- Generierte `hashCode` Funktion diskutieren

Schlechte `hashCode` kann sehr viel Performanz kosten:

- Elemente werden in Tabelle abgelegt, der Hashtabelle
- Tabelleneintrag speichert Elemente mit Hashwerten aus einem Bereich, dieser Eintrag wird Bucket genannt
- Kollision bedeutet zwei Elemente mit gleichem Hashwert, d.h. mehr als ein Element pro Bucket.

Bei jedem Zugriff wird `hashCode` berechnet, und danach der entsprechende Bucket durchsucht:

- Ein Element pro Bucket: $\mathcal{O}(1)$
- Alle Elemente in einem Bucket: $\mathcal{O}(n)$

Bucket-Größe kann sich dynamisch ändern;
hashCode von Objekten in Buckets darf sich nicht ändern

Endliche Abbildungen

Einer Menge von Schlüssel-Objekten der Klasse `K` wird jeweils genau ein Werte-Objekt der Klasse `V` zugeordnet.

Abbildung kann als zweispaltige Tabelle aufgefasst werden, wobei keine zwei Zeilen den gleichen Schlüssel haben.

Funktionalität

- Abfragen ob Schlüssel vorhanden ist
- Abfragen des Wertes eines eingetragenen Schlüssels
- Eintragen/Entfernen von Schlüssel/Wert Zuordnungen

Schlüssel bilden eine endliche Menge!

Interface `Map<K,V>`


```
interface Map<K,V>
  V        get(Object key);      // may return null
  V        remove(Object key);   // may return null
  V        put(K key, V value); // may return null
  boolean          containsKey(Object key);
  boolean          containsValue(Object value);
  Set<K>           keySet();
  Collection<V> values();
  ...
```

- Bildet Schlüssel-Objekte `K` auf Werte-Objekte `V` ab
- Schlüssel-Objekte sollten möglichst immutable sein
- Vergleiche benutzen `equals` und oft auch `hashCode`

Implementierungen von Abbildungen

- Hashtabelle speichert Paare von Schlüsseln und Werten
  Nachteile:
  - Effizienz hängt stark von `hashCode` ab
  - Einfügen kann rehashing erfordern

- Durchsuchen einer schnell durchlaufbaren Datenstruktur, z.B. Suchbäume
  Nachteile:
  - Suchen, Einfügen und Löschen meist $\mathcal{O}(\log n)$
  - Einfügen oder Löschen kann Restrukturierung erfordern

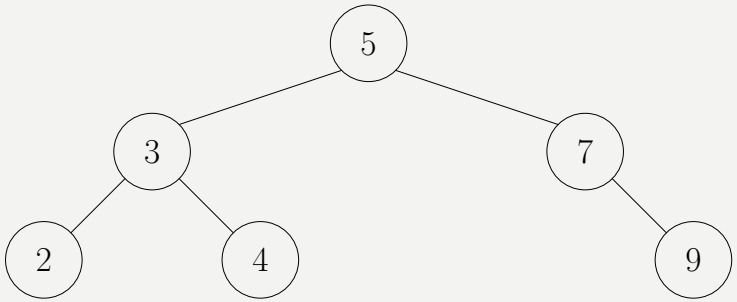
Worst-case in beiden Fällen jeweils $\mathcal{O}(n)$

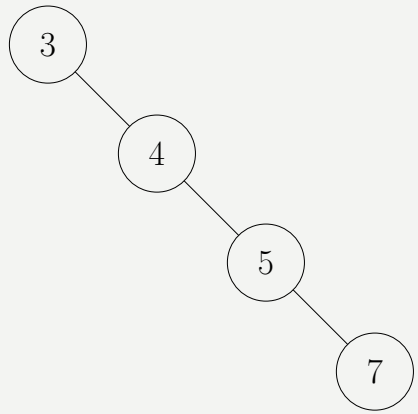
Definition:

Binärer Baum, mit total geordneten $<$ Beschriftungen, so dass für alle Knoten x gilt:

1. für jedes y im linken Teilbaum von x gilt $y < x$
2. für jedes z im rechten Teilbaum von x gilt $x < z$

Beispiele





Suche: Anzahl Vergleiche $\mathcal{O}$ (Baum-Höhe) $\leq \mathcal{O}(n)$
Balancierte Suchbäume (AVL, Red-Black) verhindern Degeneration, erfordern aber rebalancing

Interface `Comparable`

Interface für totale Ordnungen

```
interface Comparable<T> {
  int compareTo<T obj>;
}
```

Objekte von Klassen, welche diese Interface implementieren, können in Suchbäumen verwendet werden (z.B. `TreeMap`)

Es muss gelten `x.equals(y) == True` genau dann, wenn `x.compareTo(y) == 0`

Alternativ kann aber auch ein externer Vergleich benutzt werden (siehe `Comparator`) je nach Zweck des Vergleichs

Endliche Listen

Endliche Liste von Elementen gleichen Typs

- Elemente können mehrfach enthalten sein
- Reihenfolge der Elemente wichtig

Funktionalität

- Hinzufügen/Entfernen eines Elementes
- Auslesen eines Elementes an einer Position
- Ersetzen eines Elementes an einer Position
- Iteration durch alle Elemente in festgelegter Reihenfolge

Spezialisierungen

Stack Einfügen/Löschen nur am Anfang der Liste

Queue Löschen am Anfang, Einfügen am Ende

Interface `List<E>`


```
interface List<E> extends Collection<E>
  boolean contains(Object o);
  boolean isEmpty();
  boolean add(E e);
  boolean remove(Object o);
  E       set(int index, E element);
  Iterator<E>     iterator();       //respects order
  ListIterator<E> listIterator();   //respects order
  List<E>         subList(int fromIndex, int toIndex);
  ...
```

- Unterschiedliche Komplexität je nach Implementierung, Iteration gegenüber Index-Zugriff bevorzugen
- Veränderliche Objekte seltener problematisch
- Vergleiche benutzen Methode `equals`

LinkedList

Elemente in Knoten-Objekte verpackt, welche Referenzen auf Vorgänger und/oder Nachfolger merken

Vorteile
- Einfügen/Entfernen an Anfang/Ende $\mathcal{O}(1)$

Nachteile
- Direkter Index-Zugriff auf Element $\mathcal{O}(n)$

ArrayList

Elemente werden einem internen Array gespeichert

Vorteile
- Index-Zugriff $\mathcal{O}(1)$

Nachteile
- Einfügen kann Array-Kopie erfordern $\mathcal{O}(n)$
- Löschen von Elementen gibt keinen Speicher frei

Don’t

Don’t use

Arrays

Don’t use arrays!

Arrays

Arrays wegen Rückwärts-Kompatibilität problematisch:

```
Object[] objArray = new Long[1];
objArray[0] = "I don't fit";    // Runtime Exception

List<Object> objectList = new ArrayList<Long>();
// Line above does not compile
objList.add("I don't fit");
```

- Arrays ineffizienter, da Laufzeit-Typprüfungen notwendig
- `ArrayList` bessere Wahl wegen statischer Typsicherheit
- `EnumSet` anstatt Bit-Arrays: gleiche Effizienz, viel lesbarer

Quellen

- “Effective Java” (2nd Edition) von Joshua Bloch, Addison-Wesley Verlag, 2008
- Vorlesungsfolien zum Softwareentwicklungspraktikum, Wintersemester 2008/09, LMU München
- Dokumentation der Java, Platform Standard Edition 7, von Oracle
  `http://docs.oracle.com/javase/7/docs/api/`