Datenstrukturen in Java

Datenstrukturen

Datenstrukturen ermöglichen Verwaltung von / Zugriff auf Daten (hier: Objekte)

Datenstrukturen unterscheiden sich duch

- Funktionalität
- Implementierung

modulares Design: Datenstrukturen gleicher Funktionalität aber verschiedener Implementierung sollten untereinander austauschbar sein; evtl. Auswirkung auf Performanz, aber nicht auf syntaktische und semantische Korrektheit

Objektorientierung: Funktionalität durch Interface beschrieben

Beispiele

Beispiele für bereitgestellte Funktionalität:

- (endliche) Menge (Interface `Set<E>`)
- (endliche) Liste (Interface `List<E>`)
- (endliche) Abbildung (Interface `Map<K,V>`)
- ...

Beispiele für unterschiedliche Implementierung:

- Mengen: `HashSet`, `TreeSet`, ...
- Listen: `ArrayList`, `LinkedList`, `Vector`, ...
- ...

Vorsicht: Implementierung eines Interfaces kann auch zu Spezialisierung führen: `EnumSet<E extends Enum<E>>`

Endliche Mengen

endliche Mengen sind Kollektionen mit Eigenschaften

- alle Elemente sind verschieden (bzgl. üblicher Gleichheit oder abgeleitet von $<$)

Funktionalität

- Test auf Enthaltensein eines Objekts
- Hinzufügen (auch schon vorhandener Objekte)
- Löschen (auch nicht vorhandener Objekte)
- Vereinigung, Durchschnitt, ...
- (Iteration über alle Elemente der Menge)
- ...

Das Interface `Set<E>`

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

zu beachten: Typ `Object` im Argument

- lässt statisch auch z.B. Anfragen mit Argumenten nicht vom Typ `E` zu
- vermutlich historische Gründe
- Vergleiche benutzen Methode `equals`

Listen

endliche Liste ist lineare Datenstruktur; jedes (bis auf letztes) Element hat genau einen Nachfolger; jedes (bis auf erstes) Element hat genau einen Vorgänger

Funktionalität

- Einfügen, Löschen, Ersetzen (an bestimmter/beliebiger Stelle)
- Iteration durch alle Elemente
- Suchen eines Elementes

nicht unbedingt direkten Zugriff auf bestimmte Stelle wie bei Array

Spezialisierungen:

- Stack: Einfügen, Löschen nur am Anfang
- Queue: Löschen am Anfang, Einfügen am Ende

Das Interface `List<E>`

```
interface List<E> {
  ...
  boolean contains(Object o);
  boolean add(E e);
  boolean add(int i, E e);
  boolean remove(Object o);
  boolean addAll(Collection<? extends E> c);
  boolean retainAll(Collection<?> c);
  ...
}
```

beachte:

- Argumenttyp `Object` wie bei `Set<E>`
- `addAll(Collection<E>)` nicht gut, da Typkontexte Untertypbeziehungen nicht erhalten

Abbildungen

Bsp.: Zuordnungen mit endlichem Domain

1. nein: jeder Arena die darauf angemeldeten Spieler zuordnen
2. ja: pro Arena jedem Spieler eine Farbe zuordnen

(1) leicht durch Instanzvariable in Arena zu lösen; typischerweise kein Suchen nach einer bestimmten Arena nötig

(2) Spieler fungiert als Schlüssel, Farbe als Wert; typischerweise sucht man nach Farbe für gegebenen Spieler

Abbildung kann man sich als zweispaltige Tabelle vorstellen; keine zwei Zeilen mit gleicher erster Spalte (Schlüssel) vorhanden

Abbildungen – Funktionalität

geg. Menge K der Schlüssel, Menge V der Werte

- Abfragen des Werts zu einem Schlüssel k
- Eintragen einer neuen Abhängigkeit $k \mapsto v$
- Löschen eines Schlüssels (und seines Werts)
- Abfrage nach Vorhandensein eines Schlüssels
- (Iteration über alle Schlüssel-Wert-Paare)
- ...

Das Interface `Map<K,V>`

```
interface Map<K,V> {
  ...
  V get(Object k);
  V put(K k, V v);
  V remove(Object k);
  boolean containsKey(Object k);
  ...
  Set<Map.Entry<K,V>> entrySet();
  Set<K> keySet();
  Collection<V> values();
  ...
}
```

beachte:

- vorhandene Schlüssel bilden Menge, Werte jedoch nicht

Zugriff auf Objekte

unabhängig von Art der Datenstruktur besteht in einer Implementierung das Problem des Zugriffs

einfachster Fall: finde gegebenes Objekt in Datenstruktur

im Prinzip zwei Möglichkeiten

- Durchsuchen der Datenstruktur von festem Ausgangspunkt aus; Vergleiche zwischen gegebenem und vorhandenen Objekten

  erfordert schnell zu durchlaufende Stuktur $\rightsquigarrow$ Suchbäume

- direkter Sprung an eine Stelle innerhalb der Struktur, die von gegebenem Objekt abhängt (Hashing)

  Unterschied zu Array-artigen Strukturen: diese verwalten eher die Stellen, an denen Objekt abgelegt sind anstatt Objekte selbst

Suchbäume

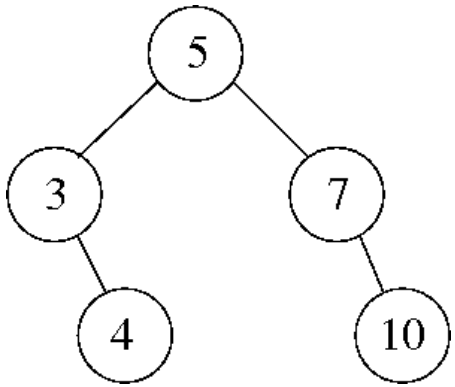
geg. Menge M mit totaler Ordung $<$

Def.: Suchbaum ist binärer M-beschrifteter Baum, sodass für alle Knoten v gilt:

- kommt u in linkem Teilbaum von v vor, so gilt $u < v$
- kommt u in rechtem Teilbaum von v vor, so gilt $u > v$

ab jetzt:

- $n = |M|$
- $m =$ Anzahl der Elemente im Baum
- $h =$ Höhe des Baums



Operationen auf Suchbäumen

- Suchen eines Objekts in $\mathcal{O}(h)$, worst-case $\mathcal{O}(m)$
- Einfügen ebenfalls
- Löschen ebenfalls
  - Blatt in $\mathcal{O}(1)$
  - innerer Knoten muss ersetzt werden durch größten in linkem oder kleinsten in rechtem Teilbaum

wichtig: Abfrage $u < v$ wird hier als $\mathcal{O}(1)$ angenommenen

in Java-Implementierung: Relation $<$ muss evtl. realisiert werden; je nach Typ der Menge M kann dies mehr oder weniger effizient sein

Balancierung

Zugriffe auf Suchbäume im worst-case nicht besser als bei Listen

besser jedoch auf balancierten Suchbäumen

Def.: Suchbaum ist balanciert, wenn sich in jedem Knoten die Höhen von linkem und rechten Teilbaum nur um Konstante unterscheiden

Operationen auf balancierten Suchbäumen:

- Suchen in $\mathcal{O}(\log m)$
- Einfügen und Löschen ebenfalls in $\mathcal{O}(\log m)$, erfordert evtl. Rebalancierung: Rechts-/Linksdrehung

Balancierte Suchbäume – Beispiele

- Red-Black Trees

  Knoten sind rot oder schwarz, gleiche Anzahl schwarzer Knoten auf jedem Pfad

  nicht balanciert laut obiger starker Definition (Höhen können sich um Faktor 2 unterscheiden)

- AVL Trees

  Knoten haben Balance-Wert: Höhenunterschied der beiden Teilbäume

Hauptunterschiede:

- Anzahl Balancierungen bei einer Operation: Red-Black $\mathcal{O}(1)$, AVL $\mathcal{O}(\log m)$
- AVL konzeptuell einfacher
- Red-Black oft in der Praxis besser

Alternativen

- Splay Trees

  Operation "splaying": bei Zugriff auf Knoten v wird dieser sukzessive zur Wurzel befördert

  Suchen, Einfügen, Löschen in amortisierter Zeit $\mathcal{O}(\log m)$; kein zusätzlicher Speicher nötig

  schneller Zugriff auf häufig verwendete Elemente;
  Anwendungen: Cache, Garbage Collector, ...

- Scapegoat Trees

  Suchen in worst-case $\mathcal{O}(\log m)$
  Einfügen / Löschen in amortisiert $\mathcal{O}(\log m)$
  ebenfalls kein zusätzlicher Speicher nötig

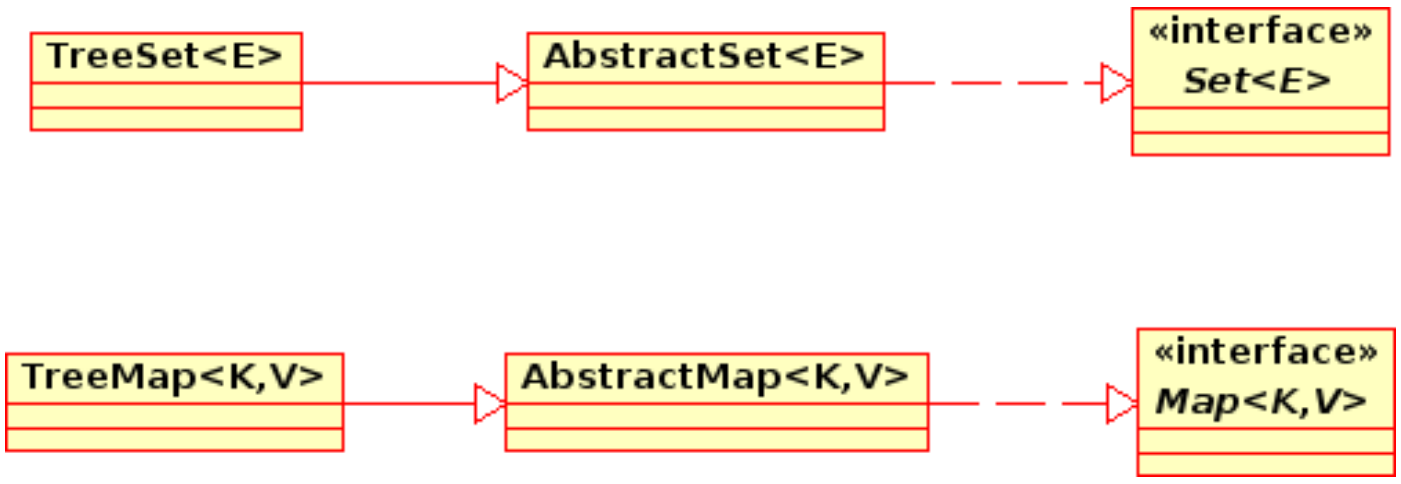
- ...

Suchbäume in Java

in Package `java.util` werden Red-Black Trees verwendet

- `TreeSet<E>` implementiert `Set<E>`
- `TreeMap<K,V>` implementiert `Map<K,V>`

genauer:



Die totale Ordnung in Suchbäumen

Interface für totale Ordnungen

```
interface Comparable<T> {
  int compareTo(T o);
}
```

Elemente von Klassen, die dieses Interface implementieren, können in Suchbäumen (also z.B. `TreeSet`, `TreeMap`) verwendet werden

Fragen:

1. wieso dann nicht gleich `TreeSet<E extends Comparable<E>>`?
2. was passiert bei `TreeSet<E>`, wenn `E` nicht `Comparable<E>` implementiert?

Comparable und Comparator

Antwort 2:

```
class TreeSet<E> {
  ...
  TreeSet() { ... }
  TreeSet(Comparator<? super E> c) { ... }
  ...
}
```

bei Mengenkonstruktion kann ein Objekt übergeben werden, welches Vergleiche auf `E` durchführt

statt `compareTo(E x)` in `E` wird `compare(T x, T y)` in `Comparator<T>` für Vergleiche benutzt

Antwort 1: `E` kann durch Implementation von `Comparable` auf eine Art total geordnet sein, für Verwaltung in Suchbäumen bietet sich aber evtl. andere totale Ordnung an

Konsistenz zwischen `compareTo` und `equals`

Def.: die durch `compareTo` bzw. `compare` definierte Ordnung auf Typ `E` ist konsistent zu `equals`, falls für alle Elemente `x`,`y` vom Typ `E` gilt:

> `x.compareTo(y)==0` bzw. `compare(x,y)` hat selben booleschen Wert wie `x.equals(y)`

beachte: `x.compareTo(null)` wirft `NullPointerException`, aber `x.equals(null)` ist `false`; `null` ist aber nicht Instanz einer Klasse

Konsistenz wichtig, denn

- Vertrag des Interfaces `Set<E>` bezieht sich auf `equals`
- Implementierung `TreeSet<E>` benutzt `compareTo` oder `compare` für Vergleiche

Hashing

zur Erinnerung

- Elemente einer Menge M werden in Tabelle abgelegt, genannt *Hashtabelle*
- Tabelleneintrag ist *Bucket*, kann mehrere Elemente halten
- Position eines Eintrags in der Tabelle ist gegeben durch Funktion $M \to \mathbb{N}$, genannt *Hashfunktion*
- *Kollision* = zwei Einträge an gleicher Position
- abzubildenende Elemente werden auch *Schlüssel* genannt

Ziel: Hashfunktion so wählen, dass Kollisionen minimiert werden und somit Zugriff in $\mathcal{O}(1)$ möglich ist

Bucket z.B. durch Liste realisiert; Zugriff linear in Listenlänge (also diese möglichst konstant im Vergleich zur Größe der Hashtabelle

Hashtabellen für Mengen und Abbildungen

Hashtabellen modellieren Mengen auf natürliche Art: Menge aller in der Tabelle enthaltenen Elemente; einzelne Buckets jeweils wieder als Mengen realisieren

Funktionalität für Mengen leicht zu realisieren

Bsp.: zur Suche von Element x, berechne Hashwert $h(x)$ von x, durchsuche Bucket an Stelle $h(x)$ nach Vorkommen von x

für Abbildungen vom Typ $f : K \to V$

- Hashfunktion h ist vom Typ $K \to \mathbb{N}$
- Bucket an Stelle $h(k)$ enthält Eintrag $(k, f(k))$

Hashfunktionen

verschiedene Methoden, jeweils mit unterschiedlicher Güte in verschiedenen Anwendungsfällen

- Extraktion: verwende lediglich Teil des Schlüssels
- Division: Abbildung auf Integer, dann modulo Tabellengröße
- Faltung: teile Schlüssel auf, verbinde Teile (z.B. Addition)
- Mitte-Quadrate: Abbildung auf Integer, Quadrierung, genügend großes Stück aus Mitte nehmen
- Radixtransformation: Abbildung auf Integer, Transformation in andere Basis, dann Division
- ...

hier keine detaillierte Diskussion der jeweiligen Güte

Dynamik einer Hashtabelle

Hashtabellen typischerweise parametrisiert durch

- initiale Kapazität: wieviele Tabelleneintrage zu Anfang
- (Schrittweite der Vergrößerungen: wieviele Einträge kommen hinzu, wenn Tabelle zu klein erscheint)
- Auslastungsfaktor: wieviele Elemente dürfen in der Tabelle höchstens vorhanden sein (in Relation zur Anzahl der Buckets), bevor diese vergrößert wird

gute Werte hängen natürlich von jeweiliger Anwendung ab

zu beachten: bei Tabellenvergrößerung werden alle eingetragenen Werte neu gehasht (z.B. per Divisionsmethode modulo neuer Tabellengröße)

- ganz andere Verteilung in der neuen Tabelle möglich
- Vergrößerungen aufwändig, also möglichst vermeiden; keine unnötigen Elemente in Tabelle, gute Parameterwahl, ...

Die Methode `hashCode` in Java

jedem Objekt hat standardmäßig einen Hashwert

```
class Object {
  ...
  int hashCode() { ... }
  ...
}
```

- Abbildung ist an sich sehr gute Hashfunktion
- Überschreiben bietet sich jedoch an (zwei an sich gleiche Objekte haben vermutlich verschiedene Hashwerte)
- noch bei Java 1.3: `hashCode` bildete lediglich Speicheradresse auf `int` ab; evtl. dynamische Änderung des Hashwerts durch Garbage Collection möglich

Verwendung von Hashtabellen

Methode `hashCode` liefert generischen Hashwert; Hashtabelle nutzt diesen, um Schlüssel auf Integer abzubilden; eigentlicher Hashwert (Position in geg. Tabelle) wird von Methoden in Hashtabellenimplementierung berechnet

wird `equals` überschrieben, so sollte man auch `hashCode` überschreiben

ebenfalls Konsistenzproblem:

- Vertrag von `Set<E>` bezieht sich auf `equals`
- Implementation `HashSet<E>` benutzt `hashCode`, um Bucket zu finden; nur innerhalb des Buckets werden Vergleiche gemacht

beim Überschreiben von `equals` zu beachten: nicht nur überladen! Argument ist `Object`

Überschreiben von `equals` und `hashCode`

folgendes muss bei `equals` beachtet werden

- Symmetrie: `x.equals(y)` hat immer selben Wert wie `y.equals(x)`
- Reflexivität: `x.equals(x)` ist immer `true` ausser wenn `x` = `null`
- Transitivität: wenn `x.equals(y)` und `y.equals(z)` beide `true` sind, dann auch `x.equals(z)`
- Konsistenz: wenn `x.equals(y)` den Wert `true` hat, dann gilt `x.hashCode()` = `y.hashCode()`
- Determiniertheit: die Werte von `equals` und `hashCode` hängen nur vom Zustand der involvierten Objekte ab

bei überschriebenem `hashCode` kann dennoch auf Standard-Hashwert zurückgegriffen werden: `System.identityHashCode(Object o)`

Die Hash-Klassen

- `HashSet<E>` implementiert Interface `Set<E>` auf Basis einer Hashtabelle
- zwei Implementierungen von `Map<K,V>` auf Basis von Hashtabellen:
  - `Hashtable<K,V>`
  - `HashMap<K,V>`

  Unterschiede: `Hashtable` ist synchronisiert, `HashMap` lässt auch `null` als Wert zu
- Varianten von `HashMap`:
  - `IdentityHashMap`: verwendet Referenzgleichheit (==) statt Objektgleichheit (`equals`)
  - `WeakHashMap`: erlaubt Freigabe von Objekten, die nur noch durch Hashtabelle referenziert werden
  - `LinkedHashSet` und `LinkedHashMap`: hält Objekte in Hashtabelle zusätzlich in doppelt verketteter Liste, um Einfügereihenfolge zu speichern