ADS Zusammenfassung

René Bernhardsgrütter 02.04.2012

1 Generics

Gewähren Typsicherheit und können für verschiedene Datentypen ohne Casts verwendet werden. Beim Erstellen der Klasse werden Platzhalter für den Typ eingesetzt, der später ersetzt wird.

Beispiele:

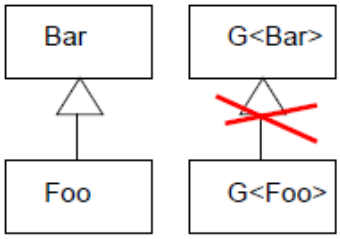
```
public class Box<T> {
        private T data;
        public T get() {…}
        public void set(T data) {…}
}
```

```
public interface Stack<E> {
        public void push(E e);
        public E pop();
}

public class ListStack<E> implements Stack<E> {
        private List<E> list = null;
        …
}

Stack<Person> personStack = new ListStack<Person>();
```

Wenn Foo von Bar geerbt hat und G ein generischer Typ ist, ist G<Foo> kein Subtyp von G<Bar>.



Superklasse aller Collections ist nicht Collection<Object> sondern heisst **Collection of unknown** und wird **als Collection<?>** geschreiben (? = Wildcard).

```
void processCollection (Collection<?> c) {
    c.add(new Integer(17));
}
```

ok?

no!
Compile time error.
c is a collection of unknown.
It could be a list of strings and we add an integer to it.

Bei der Vererbung (man will eine Subklasse einer Superklasse) muss man mit **Bounded Wildcards** arbeiten: **List<? extends Shape>**.

Auch wenn man nur Klassen will, die ein Interface implementieren, muss man dieselbe Struktur verwenden: **<E extends Comparable<E>>** (Comparable ist ein generisches Interface, das die Methode int compareTo(E element) definiert).

Man kann auch die **Lower-Bounded-Wildcard** verwenden: **List<? super Rectangle>**. Das erlaubt alle Listen, die eine **Superklasse von Rectangle** sind.

Man darf keine Generics als Array verwenden! **private E[] values;** ist nicht erlaubt.

Man kann Instanzen nicht abfragen, ob sie von einem bestimmten generischen Typ sind, da dies zur Laufzeit

```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { ... } // illegal
```

nicht mehr bekannt ist.

Man kann keine Instanzen von generischen Typen erzeugen (compilie error), da generische Typen zur Laufzeit nicht existieren => man kann den aktuellen Typ dann nicht herausfinden.

2 Stack

Man kann nur oben drauf legen und oben wieder wegenehmen (fist in last out). Implementation z. B. mit List. Typische Methoden: push (hinzufügen), pop (entfnern und ausgeben), isEmpty, peek (ausgeben, ohne es aus dem Stack zu entfernen), isFull (…). Man arbeitet so, dass man bei der Liste immer mit add(0) vorne eins hinzufügt bzw. mit remove(0) das erste wieder wegnimmt (Liste wird immer nach hinten bzw. nach vorne verschoben).

3 Queue

Objekte werden in der selben Reihenfolge entfernt, wie sie hinzugefügt wurden (first in first out). Übliche Methoden: enqueue (hinzufügen), dequeue (entfernen), isEmpty, peek (gibt das erste zurück, ohne es aus der Queue zu löschen).

4 Listen

Methoden: add(x), get(int i), remove(int i), size()…

Daten werden im Node als E data; gespeichert (Sichtbarkeit: Package). In jedem Node wird mit ListNode<E> next; ein Pointer auf den nächsten ListNode gespeichert. Wenn next==null; ist die Liste fertig (letzter Node).

Wenn die Sichtbarkeit der Daten nicht package sein soll, muss ListNode eine Innere Klasse von List sein, dann können die Variablen private sein (äussere Klassen können auf alle Werte der inneren zugreifen).

Typen:

- LinkedList: Jeder Knoten kennt das nächste Element.
- Ringlist: Das letzte Element zeigt bei next auf das erste Element (mit dummy-Element).
- Doppelt verkettete Liste: Jeder Node kennt Vorgänger und Nachfolger.
- Doppelt verkettete Ringliste: Das letzte Element zeigt bei Next auf das erste Element, das erste zeigt bei Previous auf das letzte Element.

5 Iterator

Information-Hiding: Mit dem Iterator kann man über eine Datenstruktur iterieren (traversieren, darüber gehen), ohne, dass man die Datenstruktur kennen muss. Es muss einfach von allen betroffenen Objekten das Interface Iterator implementiert werden. Methoden dessen:

- Boolean hasNext();
- E next() (liefert das nächste Element zurück)
- Void remove() (löscht das zurückgegebene Element => **Iterator speichert aktuelle Position!**)

Listen haben typischerweise eine Methode **iterator()**, die **einen** Iterator zurückgeben. **Es sind gelichzeitig mehrere Itereatoren auf eine Liste anwendbar und man kann darüber iterieren und dennoch auf die Liste zugreifen!**

Damit for (String string : strings) funktioniert, muss String das Interface **Iterable** implementieren!

6 Sortierte Listen

Daten werden vor dem Einfügen verglichen. Dazu **müssen die Daten das Interface Comparable implementieren**: **int** x.compareTo(y):

- Wenn < 0: x ist kleiner y
- Wenn =: x ist gleich y (Grösse davon)
- Wenn > 0: x ist grösser y

7 Collection

Abstrakter Datentyp für Sammlungen (Collections) von Objekten. NICHT ARRAY! Ein Framework in Java, das all diese Datenstrukturen (heute etwa 25) implementiert…

8 Rekursionen

Eine Funktion ist rekursiv, wenn sie sich selbst direkt oder indirekt aufruft.

Eine Datenstruktur heisst rekursiv, wenn sie sich selbst als Teil enthält oder mit Hilfe von sich selbst definiert ist.

Vorteile

- Möglichkeit, eine unendliche Menge durch eine endliche Aussage zur beschreiben.
- Kürzere Formulierung (in der Regel)
- Lichter verständliche Lösung (ausser es ist Shit-Code)
- Teilweise sehr effizient

Nachteile

- Weniger effizientes Laufzeitverhalten (Overhead bei Funktionsaufruf?)
- Verständnisprobleme bei Programmieranfängern
- Konstruktiver Algorithmus gewöhnungsbedürfnis

Bsp:

```
int fak(int n) {
        if (n == 0) {
                return 1;
        else {
                return n * fak (n-1);
        }
}
```

Wichtig: Man muss immer daran denken, dass die Rekursion **in keinem Fall unendlich läuft**. Sie muss immer irgendwann **terminiert werden**!

Direkte Rekursion: Die Methode ruft sich selbst wieder auf (Fakultät).
Indirekte Rekursion: Die Methode ruft eine andere Methode auf, welche dann auslöst, dass die erste Methode nochmals aufgerufen wird.

8.1 Aufwand einer Rekursion

Aufwand = Maximale Tiefe – 1;

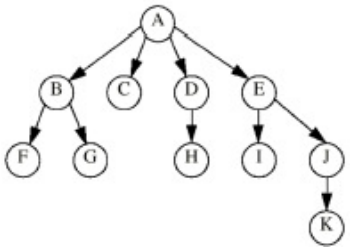
Beispiel an den Türmen von Hanoi: hanoi(3) => hanoi(2) => hanoi(1) => handoi(0) ➔ Tiefe = 4 – 1 = **3**

9 Bäume

Verzweigungen: **Knoten**
Enden: **Blätter**
Ursprung: Root / **Wurzel**
Verbindung zwischen zwei Konten oder Konten und Blatt: **Kante**

9.1 Gerichteter Baum

Root: Hat nur Ausgänge. Alle anderen Knoten: Haben genau einen Eingang und 0 bis mehrere Ausgänge.

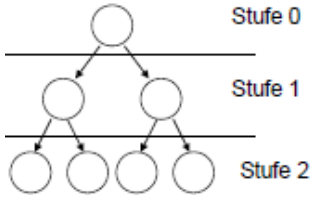


Tiefe = Wie weit der entfernteste Knoten vom Root entfernt ist (bei dem Beispiel rechts sind des 3).

Gewicht = Anzahl aller Knoten.

9.2 Binärbäume

Maximal zwei Kinder! Ein Binärbaum ist entweder leer, oder besteht aus einem Wurzelknoten und aus einem linken und einem rechten Teilbaum.

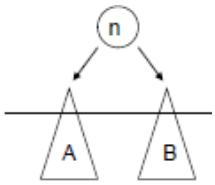


K = Tiefe; Auf jeder Stufe hat es 2^s Knoten, ein Baum hat maximal 2^{k+1} Knoten.

Traversieren = Durchlaufen eines Baumes mit einer (rekursiven) Methode.

Arten der Traversierung

Preorder, Knoten zuerst: n, A, B
Inorder, Knoten in der Mitte: A, n, B
Postorder, Knoten am Schluss: A, B, n
Levelorder: n, a_0, b_0, a_1, a_2, b_1, b_2, ..



Löschen von Knoten

Drei Fälle möglich:

1. Der Knoten hat keine Kinder => Knoten löschen
2. Der Knoten hat ein Kind
3. Der Knoten hat zwei Teilbäume

Suchen im Binärbaum

Bei einem vollen Binärbaum müssen lediglich $\log_2$ Schritte durchgeführt werden bis Element gefunden wird.

9.3 Vergleiche mit Comparable

Comparable vergleicht die Grössen der Elemente und gibt einen Integer zurück. Dazu **müssen die Daten das Interface Comparable implementieren**: **int** x.compareTo(y):

- Wenn < 0: x ist kleiner y
- Wenn =: x ist gleich y (Grösse davon)
- Wenn > 0: x ist grösser y

9.4 Ausgeglichene Bäume

Voll = Bis auf die letzte Stufe aufgefügllt.

AVL-ausgeglichen: Für jeden Knoten gilt, das dessen Tiefen der beiden Teilbäume unterscheiden sich maximal um 1 (AVL = Adelson-Velskij-Landis => Entwickler).

AVL-vollständig-ausgeglichen: Alle Teilbäume sind AVL-ausgeglichen (total gibt es einen maximalen Unterschied von 1 von dem tiefsten Blatt zum höchsten bzw. vom höchsten zum tiefsten).

Vorteile

- Einfacher zu realisieren als Gewichtsbedingung
- Degenerierung zu einer Liste ist nicht möglich
- Suchoperationen sind sehr schnell

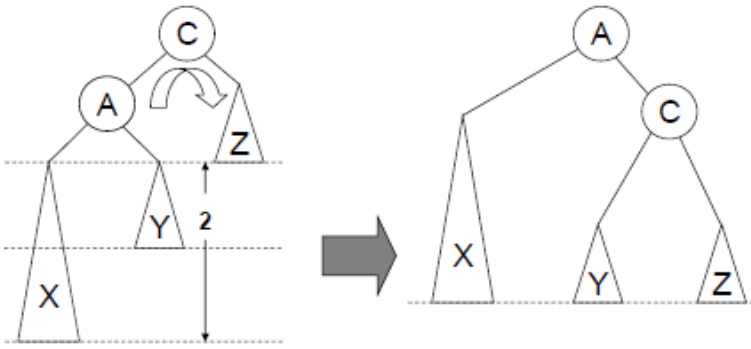
Nachteile

- Zusätzlicher Aufwand bei der Programmierung
- Einfügen und Löschen sind aufwändiger

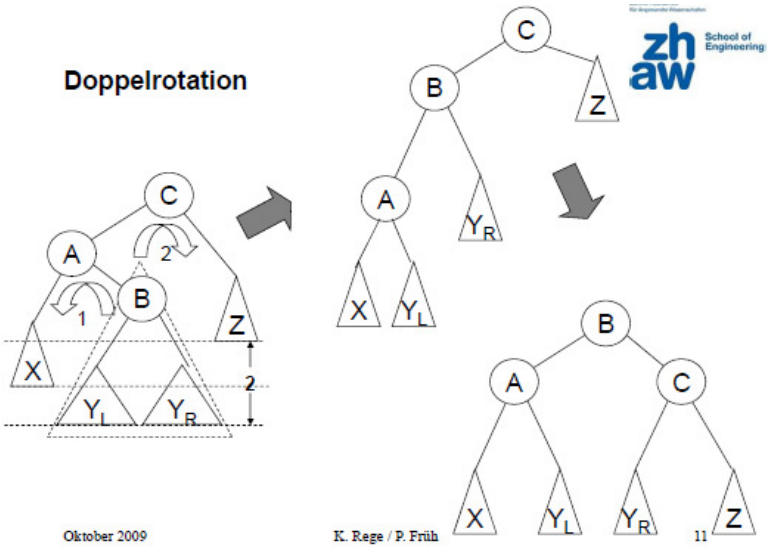
Rotationen zum Wiederherstellen der Ausgeglichenheit

Wenn man einen Knoten löscht oder hinzufügt, kann es sein, dass die oben erwähnt AVL-Ausgeglichenheit zerstört wird. Dies muss aber teilweise garantiert werden. Um dies zu korrigieren werden Rotationen eingesetzt.

Dann kann die Differenz zwischen den Blättern wieder auf maximal 1 verringert werden.

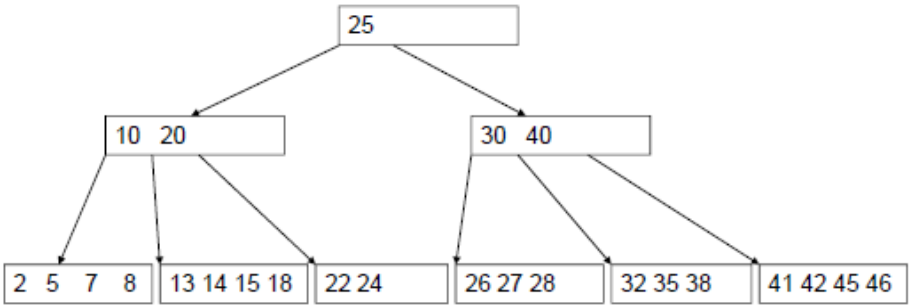


Wenn man es mit einer (Einzel-)Rotation nicht in den Griff bekommt, kann man eine **Doppelrotation** anwenden:



Ein B-Baum mit „Ordnung n“

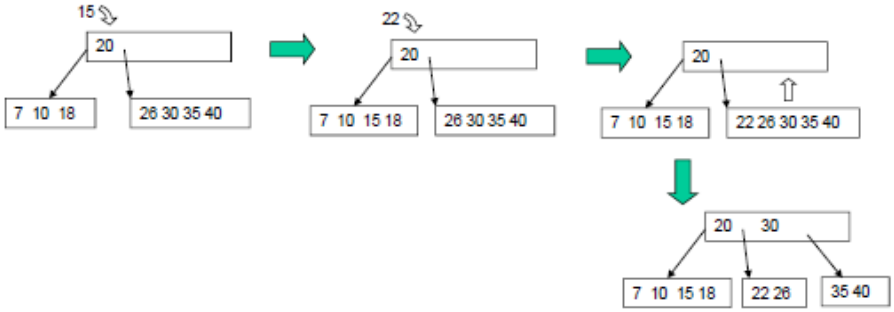
In einem B-Baum der Ordnung n enthält jeder Knoten ausser der Wurzel mindestens n und höchstens 2n Schlüssel.



Einfügen

- Es wird immer in den Blättern eingefügt.
- Einfügen innerhalb Konten, bis dieser voll ist: Überlauf ab 2n + 1
- Aufteilung des überlaufenen Knotens in zwei Knoten und „heraufziehen“ des mittleren Elements (das fünfte Rad am Wagen), dieses wird zum neuen Vaterknoten.

- Falls der Vaterknoten ebenfalls überläuft, wird dieser wieder aufgeteilt usw.



Suchen

Anzahl Zugriffe = Proportional zur Tiefe des Baumes. Tiefe des Baums selten grösser als 5 oder 6.