Datenstrukturen

Adrian Schüpbach
adrian_laurent.schuepbach@alumni.ethz.ch

Worum geht es?

- Datenstrukturen Teil 2
  - Mehr Hintergrundinfos
  - Datenstruktur-Praxis
- Templates und generische Programmierung

Map vs. Set

- Gesehen: Es gibt `TreeMap`, `TreeSet`, `HashMap`, `HashSet`,
- Unterschied Map $\leftrightarrow$ Set?

Map vs. Set

- Gesehen: Es gibt `TreeMap`, `TreeSet`, `HashMap`, `HashSet`,
- Unterschied Map $\leftrightarrow$ Set?
- Map
  - Speichert Schlüssel und zugehörigen Wert
  - Kann mehrere *gleiche Werte* enthalten (Schlüssel eindeutig)
- Set
  - Speichert nur Schlüssel
  - Jeder Schlüssel nur einmal vorhanden

Map vs. Set

Schlüssel

- Schlüssel
  - Muss eindeutig sein
  - Quasi “Index” in Datenstruktur

Map vs. Set

Schlüssel

- Schlüssel
  - Muss eindeutig sein
  - Quasi “Index” in Datenstruktur
- Schlüssel so wählen, dass oft danach gesucht werden kann
  - Suchen nach Schlüssel schnell
  - Datenstruktur nach Schlüsselsuche optimiert

Map vs. Set

Schlüssel

- Suche nach Wert möglich, heisst aber, dass *jeder* Wert angeschaut werden muss
- Beispiel: Baum mit Schlüssel und Wert (`TreeMap`), n Elemente
  - Suche nach Schlüssel
    - Pfad im Baum traversieren
    - $log_2 n$ Vergleiche
  - Suche nach Wert
    - Alle Werte anschauen
    - Weiss nicht, wo Wert ist, nicht nach Wert sortiert
    - n Vergleiche

Map vs. Set

Anwendungen für Map

- Daten organisieren
  - Wert = Personenobjekt
  - Schlüssel = Eindeutige Personalnummer

Map vs. Set

Anwendungen für Set

- Sich merken, ob man etwas schon gesehen hat
  - Habe ich Datei “bla.txt’ schon eingelesen?
  - Falls ja (in HashSet), muss ich sie nicht nochmals einlesen
  - Habe ich für Zeitpunkt “bla” schon alles berehnet?
- Ja/nein-Antwort genügt
  - Element ist in HashSet $\Rightarrow$ ja
  - Element ist *nicht* in HashSet $\Rightarrow$ nein

Templates und generische Programmierung

Templates und generische Programmierung

- Gesehen: Eigene Datenstruktur für `int` resp. `String` implementiert
- Was aber, wenn ich andere Objekte in Datenstruktur einfügen will?

Templates und generische Programmierung

- Möglichkeit 1: Überladen
  - Implementiere `add` und `remove` für alle Typen, die ich einfügen will

Templates und generische Programmierung

- Möglichkeit 1: Überladen
  - Implementiere `add` und `remove` für alle Typen, die ich einfügen will
  - Wieviele sind alle?
  - Welche?
  - Was, wenn später jemand zusätzliche Datentypen einfügen will?

Templates und generische Programmierung

- Möglichkeit 2: Datenstrukturen kennen nur `Object`
  - Implementiere `add` und `remove` für `Object`

Templates und generische Programmierung

- Möglichkeit 2: Datenstrukturen kennen nur `Object`
  - Implementiere `add` und `remove` für `Object`
  - Ursprüngliche Java-Lösung
  - Funktioniert immer
  - Problem: Muss beim Abfragen `Object` in tatsächlichen Datentyp casten
  - Keine Kontrolle, ob nur Objekte eines Typs eingefügt wurden
  - $\rightarrow$ Kann Fehler beim Casten geben

Templates und generische Programmierung

- Möglichkeit 3: Template (Vorlage) für Datenstrukturtyp erstellen
  - Konkreter Datentyp wird bei der Verwendung der Datenstruktur spezifiziert
  - Konkreter Datentyp ist *Parameter*
  - Implementiere `add` und `remove` für Typ des Parameters

Templates und generische Programmierung

- Möglichkeit 3: Template (Vorlage) für Datenstrukturtyp erstellen
  - Konkreter Datentyp wird bei der Verwendung der Datenstruktur spezifiziert
  - Konkreter Datentyp ist *Parameter*
  - Implementiere `add` und `remove` für Typ des Parameters

  - Saubere Lösung
  - Bei Deklaration wird Template parametrisiert $\Rightarrow$ fertiger Datentyp
  - Hat Kontrolle, dass nur Objekte dieses Datentyps eingefügt werden
  - Bei Abfragen kein Casten notwendig, da konkreter Datentyp bekannt ist

Klassische Datenstruktur

ListenObjekt

```
class ListeItObj {
    Element head; Element tail;
    class Element {
        Element next;   Object wert;
        Element (Object wert) {
            this.wert = wert;
        }
    }
    public void add(Object wert) {
        Element e = new Element(wert);
        if (head == null) { head = e;   tail = e; }
        else { tail.next = e;  tail = e; }
    }
    public Object getFirst() {
        return (head.wert);
    }
```

Template

ListenObjekt

```
class ListeItTemplate<T> {
    Element head; Element tail;
    class Element {
        Element next;   T wert;
        Element (T wert) {
            this.wert = wert;
        }
    }
    public void add(T wert) {
        Element e = new Element(wert);
        if (head == null) { head = e;   tail = e; }
        else { tail.next = e;  tail = e; }
    }
    public T getFirst() {
        return (head.wert);
    }
```

Benutzung der Datenstrukturen

Benutzung

```
public class Programm57 {
  public static void main(String[] args) {
    ListeItObj l = new ListeItObj();
    l.add(7);

    int z = (Integer)l.getFirst();
    System.out.println("z="+z);

    ListeItTemplate<Integer> l =
        new ListeItTemplate<Integer>();
    l.add(7);

    Integer z2 = l.getFirst();
    System.out.println("z2="+z2);
  }
}
```

Benutzung der Datenstrukturen: Fehlererkennung

Versehentlich String statt Integer hinzufügen

```
public class Programm57 {
  public static void main(String[] args) {
    ListeItObj l = new ListeItObj();
    l.add(new String("bla"));

    int z = (Integer)l.getFirst();
    System.out.println("z="+z);

    ListeItTemplate<Integer> l =
        new ListeItTemplate<Integer>();
    // Fehlermeldung des Compilers
    l.add(new String("bla"));

    Integer z2 = l.getFirst();
    System.out.println("z2="+z2);
  }
```

Templates und generic programming

Java

- Java benutzt Templates für Datenstrukturen
  - → Man sollte Datenstrukturen parametrisieren
  - Hat die Möglichkeit, ursprüngliche Implementation ohne Templates zu benutzen

- Eclipse:
  - Deklaration und Instanziierung ohne Typenparameter möglich
  - Zeile wird gelb markiert
  - Kann sich den Tip anschauen

Templates und generic programming

Java

- Eclipse: Bsp.:
  - `LinkedList l = new LinkedList()`
  - → `LinkedList` is a raw type. References to generic type `LinkedList<E>` should be parameterized
  - Aus Tip sieht man, dass `E` ein Typenparameter ist

Templates und generic programming

Java

- Eclipse: Bsp.:
  - `LinkedList l = new LinkedList()`
  - → `LinkedList` is a raw type. References to generic type `LinkedList<E>` should be parameterized
  - Aus Tip sieht man, dass `E` ein Typenparameter ist

- → Java-API anschauen

Templates und generic programming

Datenstrukturen in Java

Datenstruktur	Typ/Zweck
`LinkedList<E>`	Verkettete Liste: **E**lement
`Hashtable<K,V>`	Hashtable: **K**ey, **V**alue
`HashMap<K,V>`	Hashtable, nicht synchronisiert: **K**ey, **V**alue
`HashSet<E>`	Hashtable, speichert nur Schlüssel: **E**lement
`TreeMap<K,V>`	Baum: **K**ey, **V**alue
`TreeSet<E>`	Baum, speichert nur Schlüssel: **E**lement
`Stack<E>`	Stack/Stapel: **E**lement
`PriorityQueue<E>`	Min-Heap → Minimum finden: **E**lement

Templates und generic programming

Beispiel: `TreeMap<K,V>`: Klasse `Person`

```
class Person {
  static int naechstePersonalNummer = 0;
  int personalNummer; String vor, nach;
  Person(String vor, String nach) {
    personalNummer = naechstePersonalNummer++;
    this.vor = vor; this.nach = nach;
  }
  int getPersonalNummer() { return personalNummer;}
}
```

Templates und generic programming

Beispiel: `TreeMap<K,V>`: `main`

```
public class Programm58 {
  public static void main(String[] args) {
    TreeMap<Integer, Person> tm =
        new TreeMap<Integer,Person>();
    Person p = new Person("Marie-Sophie", "LeLac");
    tm.put(p.getPersonalNummer(), p);
    p = tm.get(new Integer(0));
    if (p != null) {
      System.out.println("Personalnummer " + 0
          + ": " + p.vorname + ", " + p.nachname);
    } else{
      System.out.println("Gibt es nicht");
    }
  }
}
```

Templates und generic programming

Beispiel: TreeMap<K,V>: main

```
public class Programm58 {
  public static void main(String[] args) {
    TreeMap<Integer, Person> tm =
        new TreeMap<Integer,Person>();
    HashMap<String,Person> hm =
        new HashMap<String,Person>(253);
    Person p = new Person("Marie-Sophie", "LeLac");
    tm.put(p.getPersonalNummer(), p);
    hm.put(p.nachname, p);
    p = hm.get("LeLac");
    if (p != null) {
      System.out.println("Personalnummer " + 0
          + ": " + p.vorname + ", " + p.nachname);
    } else{
      System.out.println("Gibt es nicht");
    }
```

Gleichheit

Gleichheit

- Was heisst ”gleich”?

Gleichheit

- Was heisst ”gleich”?

- Gesehen: `==`, `!=`
- Auch gesehen: `.equals()`

Gleichheit

- Was heisst ”gleich”?

- Gesehen: `==`, `!=`
- Auch gesehen: `.equals()`
- Was ist der Unterschied?
- Warum ist es relevant?

Gleichheit

- == testet, ob zwei Objekte gleich sind

Gleichheit

- == testet, ob zwei Objekte gleich sind
  - `int z, v; z == y`: Wert von z gleich Wert von v

Gleichheit

- == testet, ob zwei Objekte gleich sind
  - `int z, v; z == y`: Wert von z gleich Wert von v
  - `int[] a; int[] b; b = a; a == b`:
    Zeiger auf Array gleich, nicht Inhalt zweier Arrays

Gleichheit

- == testet, ob zwei Objekte gleich sind
  - `int z, v; z == y`: Wert von z gleich Wert von v
  - `int[] a; int[] b; b = a; a == b`:
    Zeiger auf Array gleich, nicht Inhalt zweier Arrays
  - `String a = new String("Hallo");`
    `String b = a; a == b;`:
    Zeiger a und b auf Hallo sind gleich

Gleichheit

- == testet, ob zwei Objekte gleich sind
  - `int z, v; z == y`: Wert von z gleich Wert von v
  - `int[] a; int[] b; b = a; a == b`:
    Zeiger auf Array gleich, nicht Inhalt zweier Arrays
  - `String a = new String("Hallo");`
    `String b = a; a == b;`:
    Zeiger a und b auf Hallo sind gleich
  - `String a = new String("bla");`
    `String b = new String("bla");`
    `a == b`:
    → `false`, ist **nicht das gleiche Objekt**

Gleichheit

- `.equals()` testet, ob **Inhalt** zweier Objekte gleich ist

Gleichheit

- `.equals()` testet, ob **Inhalt** zweier Objekte gleich ist
  - ```
    String a = new String("bla");
    String b = new String("bla");
    ```
    `a == b`:
    → `false`, ist **nicht das gleiche Objekt**
    ```

Gleichheit

- `.equals()` testet, ob **Inhalt** zweier Objekte gleich ist
  - `String a = new String("bla");`
    `String b = new String("bla");`
    `a == b`:
    → `false`, ist **nicht das gleiche Objekt**
  - `String a = new String("bla");`
    `String b = new String("bla");`
    `a.equals(b)`:
    → `true`, ist gleicher Text

Vergleiche

- Für Sortierung relevant, ob Objekt, gleich ist, vorher oder nachher in Reihenfolge kommt
- `int compareTo(T o)`
  - Methode gehört zu Objekt
  - Parameter nimmt anderes Objekt (der gleichen “Sorte”/Klasse)
  - Vergleicht das andere Objekt mit sich selber
  - Resultate
    - 0, wenn Objekte gleich sind
    - 1, wenn ich “grösser” als Parameter-Objekt bin
    - -1, wenn ich “kleiner” als Parameter-Objekt bin
- $\rightarrow$ Methode implementieren, falls Objekte natürliche Ordnung haben
  - „, und sich somit selbstständig vergleichen können

Vergleiche

- Für Sortierung relevant, ob Objekt, gleich ist, vorher oder nachher in Reihenfolge kommt
- `int compare(T o1, T o2)`
  - Separate Klasse, die Vergleich macht
  - Methode gehört zu separater Klasse
  - Methode nimmt zwei Objekte
  - Vergleicht die Objekte miteinander
  - Resultate
    - 0, wenn Objekte gleich sind
    - 1, wenn o1 “grösser” als o2 ist
    - -1, wenn o1 “kleiner” als o2 ist
- → Methode implementieren, falls keine natürliche Ordnung
- Externe “Ordnung” herstellen

Vergleiche

Vergleiche

- Warum muss ich das wissen?

Vergleiche

- Warum muss ich das wissen?
- Datenstrukturen benutzen `compareTo()`, um Objekte zu sortieren
- Sortieralgorithmen benutzen `compare()`, um Objekte zu sortieren
- → Wenn natürliche Ordnung gegeben: `compareTo()` super!
- Manchmal notwendig, externe Klasse mit `compare()` zu erzeugen und Instanz dem Sortieralgorithmus zu übergeben

Vergleiche

`equals()`

- `equals()` kann man einfach implementieren
- Idealerweise mit richtigem Typ
- Wenn Datenstruktur *ohne* Typenparameter verwendet wird, `equals()` mit `Object` implementieren

`equals()`

```
boolean equals(Object o) {
  return(zahl == ((MeinTyp)o).zahl);
}
```

Vergleiche

Beispiel: `equals()`

```
class Person {
  static int naechstePersonalNummer = 0;
  int personalNummer; String vor, nach;
  Person(String vor, String nach) {
    personalNummer = naechstePersonalNummer++;
    this.vor = vor; this.nach = nach;
  }
  int getPersonalNummer() { return personalNummer;}

  boolean equals(Person p) {
    return (p.getPersonalNummer()==personalNummer);
  }
}
```

Vergleiche

`compareTo()`

- `compareTo()`: Klasse muss `Comparable` implementieren
  - `class Person implements Comparable<Person>`
- Dann Methode `compareTo()` einfach implementieren
- Idealerweise mit richtigem Typ

Vergleiche

Beispiel: `compareTo()`:

```
class Person implements Comparable<Person> {
  static int naechstePersonalNummer = 0;
  int personalNummer; String vor, nach;
  Person(String vor, String nach) {
    personalNummer = naechstePersonalNummer++;
    this.vor = vor; this.nach = nach;
  }
  int getPersonalNummer() { return personalNummer;}
  public int compareTo(Person p) {
   if (p.getPersonalNummer() < personalNummer) {
     return (1);
   } else if (p.getPersonalNummer()>personalNummer){
     return (-1);
   } else {
     return (0);
   }
```

Iteratoren

Iteratoren

- Datenstrukturen helfen, Elemente schnell zu suchen
- Manchmal nötig, alle Elemente anzuschauen
  - Will nicht nach allen suchen, um alle zu sehen
  - Habe ev. keine Liste aller vorhandener Schlüssel
  - Will direkte Möglichkeit, alle nacheinander abzuarbeiten

Iteratoren

- Datenstrukturen bieten immer die Möglichkeit, auf alle Elemente zuzugreifen
- Datenstrukturen haben *Iteratoren*
  - Mechanismus, der ein Element nach dem nächsten zurückgibt
  - Reihenfolge je nach Datenstruktur
    - `TreeMap`, `TreeSet` sortiert
    - `LinkedList` in Einfügereihenfolge
    - I.A. nicht sortiert

Iteratoren

- Zugriff auf alle Elemente: `for`-Schleife
- Methode `hasNext()` prüft, ob es weiteres Element gibt
- Methode `next()` gibt nächstes Element zurück

Iteratoren

- Klasse `Iterator` beinhaltet `hasNext()` und `next()`
- Methode `iterator()` der Datenstruktur gibt Instanz von `Iterator` zurück
- Idealerweise konkreten Typ benutzen → Template mit Typ parametrisieren
  - `Iterator<String>` statt nur `Iterator`, wenn Datenstruktur `<String>` beinhaltet

Iteratoren

Set

- Set-Datenstruktur:
  - Hat nur Schlüssel als Daten
  - Direkt `iterator()` aufrufen

Iteratoren

Über LinkedList iterieren

```
public class Programm53 {
  public static void main(String[] args) {
    LinkedList<Integer> ll =
      new LinkedList<Integer>();
    ll.add(5);
    ll.add(10);
    ll.add(25);
    ll.add(30);

    System.out.println("LinkedList:");
    for (Iterator<Integer> it = ll.iterator();
                              it.hasNext(); ){
      int z = it.next();
      System.out.println(z);
    }
  }
```

Iteratoren

Über TreeSet iterieren

```
public class Programm54 {
  public static void main(String[] args) {
    TreeSet<Integer> tm =
        new TreeSet<Integer>();
    tm.add(5);
    tm.add(10);
    tm.add(25);
    tm.add(30);

    System.out.println("TreeSet:");
    for (Iterator<Integer> it = tm.iterator();
                            it.hasNext(); ){
      int z = it.next();
      System.out.println(z);
    }
  }
```

Iteratoren

Über HashSet iterieren

```
public class Programm55 {
  public static void main(String[] args) {
    HashSet<Integer> hs =
        new HashSet<Integer>();
    hs.add(5);
    hs.add(10);
    hs.add(25);
    hs.add(30);

    System.out.println("HashSet:");
    for (Iterator<Integer> it = hs.iterator();
                            it.hasNext(); ){
      int z = it.next();
      System.out.println(z);
    }
  }
```

Iteratoren

Map

- Map-Datenstruktur:
  - Hat Schlüssel *und* Werte
  - Muss entscheiden, ob Iterator über Schlüssel oder Iterator über Werte gewünscht ist
    - Schlüssel: `keySet()`
    - Werte: `values()`
  - `iterator()` auf `keySet()` oder `values()` aufrufen

Iteratoren

Über TreeMap-Schlüssel iterieren

```
public class Programm59 {
  public static void main(String[] args) {
    TreeMap<Integer,String> tms =
        new TreeMap<Integer,String>();
    tms.put(17, "bla17");
    tms.put(2, "bla2");
    tms.put(99, "bla99");
    tms.put(33, "bla33");

    for (Iterator<Integer> it =
        tms.keySet().iterator(); it.hasNext(); ) {
      Integer z = it.next();
      System.out.println("Schluessel: " + z
                + ", Wert = " + tms.get(z));
    }
  }
```

Iteratoren

Über TreeMap-Werte iterieren

```
public class Programm60 {
  public static void main(String[] args) {
    TreeMap<Integer,String> tms =
        new TreeMap<Integer,String>();
    tms.put(17, "bla17");
    tms.put(2, "bla2");
    tms.put(99, "bla99");
    tms.put(33, "bla33");

    for (Iterator<String> it =
        tms.values().iterator(); it.hasNext(); ) {
      String s2 = it.next();
      System.out.println("Naechster Wert: " + s2);
    }
  }
}
```

Iteratoren

- Iteratoren iterieren über aktuelle Daten
- Datenstruktur darf in dieser Zeit nicht verändert werden
  - Gäbe Durcheinander

Iteratoren

- Iteratoren iterieren über aktuelle Daten
- Datenstruktur darf in dieser Zeit nicht verändert werden
  - Gäbe Durcheinander
- Java überprüft, dass Datenstruktur nicht verändert wird
- Falls doch...

Iteratoren

- Iteratoren iterieren über aktuelle Daten
- Datenstruktur darf in dieser Zeit nicht verändert werden
  - Gäbe Durcheinander
- Java überprüft, dass Datenstruktur nicht verändert wird
- Falls doch...
  - → ConcurrentModificationException

Iteratoren

“Algorithmus”

1. Datenstruktur verändern
2. Neuen Iterator erzeugen
3. Iterieren
4. Datenstruktur verändern
5. **Neuen** Iterator erzeugen
6. Von vorne durchiterieren
7. ...

Ziel für Projekt

Ziel für Projekt

- Zeigen, dass man mit Datenstrukturen umgehen kann
  - Datenstrukturen erzeugen
  - Konkrete Typenparameter benutzen
  - Datenstrukturen füllen, durchsuchen, modifizieren
  - Durch Datenstrukturen iterieren
- Zeigen, warum *diese* (nicht andere) Datenstruktur gewählt
  - Alle haben `.put()` und `.get()`
  - Warum ist meine Wahl gut?
  - Was hat man sich überlegt?
  - Wissen, was passieren würde, wenn man eine andere Datenstruktur wählt