SMART INDUSTRY CAMPUS

Weitere Datenstrukturen

SMART INDUSTRY CAMPUS

Was sind Datenstrukturen?

- dienen zur Speicherung und Verwaltung von Daten
- dazu zählen:
  - ArrayList
  - HashMap
  - HashSet
  - TreeMap
  - TreeSet
  - ...

Unterscheiden sich in der Art und Weise, wie sie Daten speichern und wie darauf zugegriffen wird

SMART INDUSTRY CAMPUS

Maps

SMART INDUSTRY CAMPUS

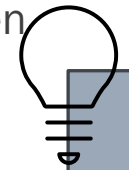
Map

- ist ein Interface in Java, welches Daten als Schlüssel-Wert-Paar speichert
- Beispiel:
  - Wir haben ein Telefonbuch
  - Hinter jedem Namen steht eine Telefonnummer
  - Nun wollen wir bspw. „Alice“ anrufen
  - Wir suchen nach Alice und finden ihre Telefonnummer. Nun können wir sie anrufen
- Zum Speichern des Namens und der Nummer kann man Maps nutzen

SMART INDUSTRY CAMPUS

HashMap<K, V>

- implementiert das Interface Map
- eine HashMap speichert Schlüssel-Wert-Paare
  - K (= key) steht für Schlüssel
  - V (= value) steht für Wert
- jeder Schlüssel ist eindeutig, d.h. darf nur einmal in der Map vorhanden sein
  - Fügt man doch zwei Schlüssel hinzu, wird der Wert überschrieben
  - Werte dürfen sich wiederholen
- interne Speicherung basiert auf Hashing
  - Dadurch ist eine schnelle Zugriffszeit möglich



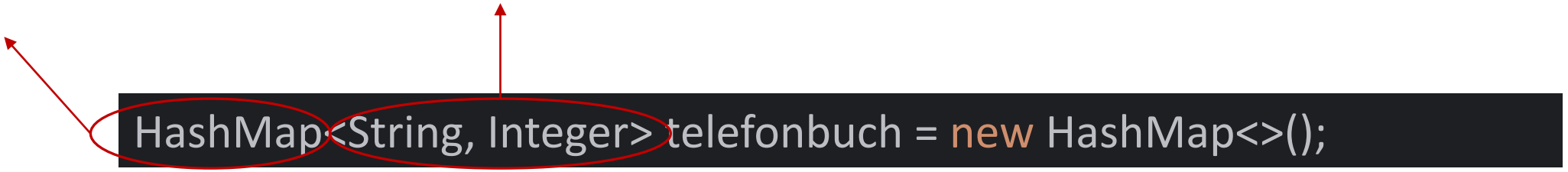
Hashing:
Prozess zur schnellen Datenlokalisierung in einer Datenstruktur

HashMap<K, V> - Initialisierung

– Beispiel Telefonbuch:

Datenstruktur
HashMap<**K**,**V**>

Key ist vom Datentyp String
Value ist vom Datentyp Integer



HashMap<K, V> - Methoden

- put(K key, V value):
  - Fügt der Map Werte hinzu

```
HashMap<String, Integer> telefonbuch = new HashMap<>();
telefonbuch.put("Alice", 12345);
telefonbuch.put("Bob", 67890);
```

Die Einfügereihenfolge wird nicht behalten!

https://javarush.com/de/groups/posts/de.2496.detaillierte-analyse-der-hashmap-klasse

Alle behandelten Methoden sind implementierte Methoden des Map Interfaces

HashMap<K, V> - Methoden

- get(Object key):
  - Holt sich den Wert (value) anhand des Keys

```
HashMap<String, Integer> telefonbuch = new HashMap<>();
telefonbuch.put("Alice", 12345);
telefonbuch.put("Bob", 67890);
System.out.println(telefonbuch.get("Alice")); // Ausgabe: 12345
```

HashMap<K, V>

- clear():
  - Entfernt alle Elemente der Map
- boolean isEmpty():
  - Prüft, ob die Map leer ist
- int size():
  - Gibt die Anzahl der Datenpaare an

HashMap<K, V>

- boolean containsKey(Object key):
  - Überprüft, ob übergebener Schlüssel (Key) in der Map enthalten ist

```
HashMap<String, Integer> telefonbuch = new HashMap<>();
telefonbuch.put("Alice", 12345);
telefonbuch.put("Bob", 67890);
System.out.println(telefonbuch.containsKey("Alice")); // Ausgabe: true
```

SMART INDUSTRY CAMPUS

HashMap<K, V>

- boolean containsValue(Object value):
  - Überprüft, ob übergebener Wert (value) in der Map enthalten ist

```
HashMap<String, Integer> telefonbuch = new HashMap<>();
telefonbuch.put("Alice", 12345);
telefonbuch.put("Bob", 67890);
System.out.println(telefonbuch.containsValue(234576)); // Ausgabe: false
```

HashMap<K, V>

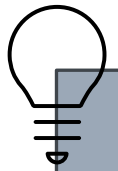
Für Schleifen nutzbar, wenn sowohl keys als auch values notwendig sind

SMART INDUSTRY CAMPUS

- Set entrySet():
  - Gibt ein Set an Schlüssel und Werten zurück, die in der Map enthalten sind
- Object getKey()
  - Gibt den Schlüssel eines Map-Objektes zurück
- Object getValue()
  - Gibt den Wert eines Map-Objektes zurück

```
HashMap<String, Integer> telefonbuch = new HashMap<>();
telefonbuch.put("Alice", 12345);
telefonbuch.put("Bob", 67890);
System.out.println(telefonbuch.entrySet());
// Ausgabe: [Bob=67890, Alice=12345]
```

HashMap<K, V>



Für Schleifen nutzbar, wenn man nur Schlüssel braucht

- Set keySet():
  - Gibt ein Set zurück, das alle Schlüssel enhält

```
HashMap<String, Integer> telefonbuch = new HashMap<>();
telefonbuch.put("Alice", 12345);
telefonbuch.put("Bob", 67890);
System.out.println(telefonbuch.keySet());
// Ausgabe: [Bob, Alice]
```

HashMap<K, V>

Für Schleifen nutzbar, wenn man nur Values braucht

- Collection values():
  - Gibt eine Collection mit allen Values zurück

```
HashMap<String, Integer> telefonbuch = new HashMap<>();
telefonbuch.put("Alice", 12345);
telefonbuch.put("Bob", 67890);
System.out.println(telefonbuch.values());
// Ausgabe: [67890, 12345]
```

SMART INDUSTRY CAMPUS

HashMap<K, V>

- V remove(Object k) / boolean remove(Object k, Value v)
  - Löscht Daten nur nach Schlüssel
  - Löscht Daten nach Schlüssel und Wert

```
HashMap<String, Integer> telefonbuch = new HashMap<>();
telefonbuch.put("Alice", 12345);
telefonbuch.put("Bob", 67890);
System.out.println(telefonbuch.remove("Alice"));
System.out.println(telefonbuch.remove("Bob", 67890));
// Ausgabe:     12345
                true
```

SMART INDUSTRY CAMPUS

HashMap<K, V>

- replace(Object k, Object value)
  - Ersetzt den Wert des Schlüssels k
- replace(Object k, Object oldValue, Object newValue)
  - Ersetzt einen Wert (=value) mit einem neuen Wert, welches mit dem Schlüssel verknüpft ist

```
HashMap<String, Integer> telefonbuch = new HashMap<>();
telefonbuch.put("Alice", 12345);
telefonbuch.put("Bob", 67890);
System.out.println(telefonbuch.replace("Bob", 121246));
System.out.println(telefonbuch);
// Ausgabe:
                    67890
                    {Bob=121246, Alice = 12345}
```

SMART INDUSTRY CAMPUS

Aufgabe

Implementiere eine HashMap, die Städte als Schlüssel und deren Einwohnerzahl als Wert speichert. Füge einige Städte hinzu und gib die Einwohnerzahl einer bestimmten Stadt aus.

Gebe alle Elemente der Map aus.

Tipp: entrySet() gibt einen Entry zurück

```
for (Map.Entry<String, Integer> eintrag: staedte.entrySet()) {
    System.out.println(eintrag.getKey() + ": " + eintrag.getValue());
}
```

Achte auf die Reihenfolge der Ausgabe.
Die Reihenfolge folgt keiner festen Regel.

SMART INDUSTRY CAMPUS

TreeMap<K, V>

- eine TreeMap speichert Schlüssel-Wert-Paare **in aufsteigender Reihenfolge**
  - Die hinzugefügten Elemente (Strings, Wrapperklassen,...) müssen entweder comparable sein!
  - Oder man muss dem Konstruktor ein Comparator übergeben
- jeder Schlüssel ist eindeutig, d.h. darf nur einmal in der Map vorhanden sein
  - Werte dürfen sich wiederholen
- interne Speicherung basiert auf einem Red-Black-Tree
- hat die gleichen Methoden wie HashMap plus einige mehr!

Red-Black-Tree ist eine Datenstruktur, die einen binären Suchbaum darstellt.

SMART INDUSTRY CAMPUS

Aufgabe

Implementiere eine TreeMap, die eine sortierte Liste von Kundename und ihren Artikelname speichert.

Gebe die Map aus.

Achte auf die Reihenfolge der Ausgabe!

SMART INDUSTRY CAMPUS

Sets

SMART INDUSTRY CAMPUS

Was sind Sets?

- ist ein Interface in Java, welches das Hinzufügen doppelter Elemente einschränkt
  - Somit können nur einzigartige Werte gespeichert werden
- Beispiel:
  - Wir haben ein Event und wollen alle Leute des Unternehmens einladen
  - Eine Einladung soll nicht an eine Person mehrmals gehen
  - Falls wir in den Verteiler doch zweimal dieselbe Person eingegeben haben, wird das ignoriert

SMART INDUSTRY CAMPUS

HashSet<E>

- jeder Wert in dem Set ist eindeutig
  - Keine Werte können doppelt vorkommen

- Initialisierung:

```
HashSet<String> eventInvitation = new HashSet<>();
```

HashSet<E> - Methoden

- clear():
  - Entfernt alle Elemente eines Sets
- boolean isEmpty():
  - Überprüft, ob das Set leer ist
- int size():
  - Gibt die Anzahl der Elemente zurück

HashSet<E> - Methoden

- add(E e):
  - Fügt dem Set ein Element hinzu

```
HashSet<String> eventInvitation = new HashSet<>();
eventInvitation.add("Anita");
```

HashSet<E> - Methoden

- boolean contains(Object e):
  - Überprüft, ob in dem Set das übergebene Objekt enthalten ist

```
HashSet<String> eventInvitation = new HashSet<>();
eventInvitation.add("Anita");
System.out.println(eventInvitation.contains("Anita"));          //Ausgabe: true
```

HashSet<E> - Methoden

- boolean remove(Object e):
  - Löscht das übergebene Objekt aus dem Set

```
HashSet<String> eventInvitation = new HashSet<>();
eventInvitation.add("Anita");
eventInvitation.add("Lukas");
System.out.println(eventInvitation.remove("Anita"));
 System.out.println(eventInvitation);            //Ausgabe: [Lukas]
```

SMART INDUSTRY CAMPUS

HashSet<E> - Methoden

- Iterator iterator():
  - Gibt einen Iterator, der über die Elemente des Sets iteriert

```
HashSet<String> eventInvitation = new HashSet<>();
eventInvitation.add("Anita");
eventInvitation.add("Lukas");

Iterator<String> iterator = eventInvitation.iterator();
while(iterator.hasNext()){
   System.out.println(iterator.next());
}
```

SMART INDUSTRY CAMPUS

Aufgabe

Verwende ein HashSet, um eine Liste von Namen zu speichern und zu überprüfen, ob ein bestimmter Name bereits existiert.

TreeSet<E>

- ein TreeSet speichert eine Menge in sortierter Reihenfolge
  - Wieder nach Red-Black-Tree sortiert
- **Beispiel:**
  - **Eine Rangliste von Spielern basierend auf Punkten**

```
TreeSet<Integer> rangliste = new TreeSet<>();
rangliste.add(1500);
rangliste.add(1800);
rangliste.add(1600);
System.out.println(rangliste);
// Ausgabe: [1500, 1600, 1800]
```

SMART INDUSTRY CAMPUS

Aufgabe

Erstelle ein TreeSet, das eine sortierte Liste von Noten speichert und überprüft, welche die höchste und niedrigste Note ist.

Vergleiche

Struktur	Ordnung	Duplikate erlaubt
HashMap	Nein	Schlüssel einzigartig
HashSet	Nein	Keine doppelten Werte
TreeSet	Ja	Keine doppelten Werte
TreeMap	Ja	Schlüssel einzigartig