Datenstrukturen

PROGRAMMKONSTRUKTION

Queue

Prinzip: FIFO

Methoden:

```
boolean empty()     // überprüft, ob die Queue leer ist
boolean offer(e)    // fügt e hinten ein, true bei Erfolg, false bei Fehler
add(e)              // wie offer, aber Exception bei Fehler
poll()              // entfernt erstes Element, gibt es zurück
peek()              // gibt erstes Element zurück ohne es zu entfernen (null bei leerer Queue)
element()           // wie peek(), aber Exception bei leerer Queue
remove()            // entfernt erstes Element ohne es zurückzugeben
```

Klassen: LinkedList

Anwendungsfälle: jeder Eintrag wird genau ein Mal verwendet, kein direkter Zugriff auf „bestimmten“ Eintrag

Implementierung: `Queue<String> queue = new LinkedList<>();`

Deque

Prinzip: Queue-Operationen an beiden Enden, **eigentlich LIFO**

Methoden: …First oder …Last muss angefügt werden

```
boolean empty()                       // überprüft, ob die Deque leer ist
boolean offerFirst(e) / offerLast(e)  // wie offer bei Queue, aber an beiden Enden möglich
getFirst() / getLast()                // wie element bei Queue, Exception bei leerer Queue
peekFirst() / peekLast()              // wie peek bei Queue, …
…
```

Klassen: LinkedList

Anwendungsfälle: jeder Eintrag wird genau ein Mal verwendet, kein direkter Zugriff auf „bestimmten“ Eintrag

Implementierung: `Deque<String> deque = new LinkedList<>();`

Stack

Prinzip: LIFO, Kellerspeicher

Methoden:

```
boolean empty()     // überprüft, ob der Stack leer ist
push(e)             // fügt e vorne bzw. „oben am Stapel“ ein
pop()               // das oberste Element vom Stack, Exception bei Fehler
peek()              // gibt das oberste Element zurück ohne es zu entfernen (Exception bei Fehler)
int search(e)       // sucht im Stack nach dem obersten Element = e und gibt die Distanz von der
                       Spitze zurück (1 bei oberster Position…)
```

Klassen: Stack

Implementierung: `Stack<String> stack = new Stack<>();`

Map

Prinzip: Assoziativer Speicher, verbindet einen eindeutigen Schlüssel „Key“ mit einem Wert „Value“
→ Mapping

Methoden:

```
boolean isEmpty()             // überprüft, ob die Map leer ist
boolean containsKey(key)      // überprüft, ob ein bestimmter Key in der Map vorhanden ist
boolean containsValue(value)  // überprüft, ob ein bestimmter Value in der Map vorhanden ist
put(key, value)               // fügt ein Key-Wert-Paar in die Map ein
                              // falls Key bereits vorhanden, wird der Value übeschrieben
get(key)                      // liest den mit Key assoziierten Wert
int size()                    // gibt die Größe der Map zurück
remove(key)                   // entfernt einen Key aus der Map und den zugehörigen Wert
toString()                    // liefert eine Zeichenkette, die die HashMap repräsentiert
```

Klassen:

HashMap

- viele Elemente werden unsortiert gespeichert
- Elemente werden über Key schnell wieder verfügbar gemacht
- Internes Hashing-Verfahren ist schnell
- Sortierung der Keys nach gegebenem Kriterium nicht möglich

TreeMap

- sortiert Elemente eines Assoziativspeichers nach Schlüsseln
- bietet Zugriff auf das kleinste oder größte Element mit Methoden wie firstKey(), lastKey()
- etwas langsamer im Zugriff
- sortiert die Elemente in einen internen Binärbaum ein
- Keys müssen sich in eine Ordnung bringen lassen, wozu Vorbereitung nötig ist

Anwendungsfälle: jeder Eintrag wird wiederholt (oder nie) verwendet, man muss „bestimmten“ Eintrag adressieren

Implementierung:
```
Map<Integer, String> map = new HashMap<>();   // Integer = Key-Datentyp
Map<Integer, String> map = new TreeMap<>();   // String = Value-Datentyp
```