Algorithmen und Datenstrukturen

Vorlesung #03 – Einführung in Java Teil 3

Benjamin Blankertz

Lehrstuhl für Neurotechnologie, TU Berlin

benjamin.blankertz@tu-berlin.de

02 · Mai · 2023

Themen der heutigen Vorlesung

- Javadoc Kommentare
- *Design by Contract*
- Behandlung von Ausnahmen (*exceptions*)
- JUnit Tests
- Die Schnittstellen `Comparable` und `Comparator`
- Vorrangwarteschlange (*priority queue*, Prioritätenschlange)
- indizierte Vorrangwarteschlange

Ein Kommentar zu Kommentaren

- Kommentare tragen nichts zum Ablauf eines Programmes bei.
- Sie sind dennoch extrem wichtig.
- Kommentieren sollte man möglichst direkt beim Programmieren.
- Nur beim Programmieren in der Vorlesung dürfen die Kommentare weggelassen werden :) bzw. werden mündlich gegeben.

Standardisierte Javadoc Kommentare

- Kommentare im **Javadoc** Format können automatisch in eine API im HTML Format übersetzt werden.
- Kommentare zu Klassen und Methoden mit `/**` und `*/` einklammern, also mit doppeltem Stern in der Eröffnung.
- Elemente durch Tags, die mit `@` beginnen hervorheben, siehe Tabelle unten.
- HTML Befehle wie `<b>`, `</b>`, `<i>`, `</i>` und `<p>` können verwendet werden.
- Siehe `https://docs.oracle.com/javase/8/docs/technotes/tools/unix/javadoc.html`

Tag & Parameter	Usage
@code *literal*	Formatiert Text im Code Font
@link *reference*	Erzeugt einen Link auf eine Klasse, Interface oder Methode
@author *name*	Name des Autors
@param *name description*	Beschreibt einen Parameter der Methode.
@return *description*	Beschreibt den Rückgabewert.
@throws *classname description*	Beschreibt eine Ausnahme, die die Methode werfen kann.

Beispiel für Doc Kommentare (Klasse)

```java
/**
 * The {@code Board} class stores the state of a 9x9 <b>Sudoku</b> grid.
 * It provides functionality for Sudoku solvers. A <i>(Priority) Queue</i> can store
 * a candidate list for each cell. It employs the class {@link CandidateList}, which implements
 * the {@link Comparable} interface based on the length of the candidate list, providing a fast
 * access of the cell with the smallest number of possible candidates.
 * <p>
 * A <b>Sudoku Solver</b> based on <i>backtracking</i> is provided in the class {@link SudokuSolver}.
 * </p>
 *
 * @author Benjamin Blankertz
 */

public class Board {
    protected Queue<CandidateList> candidates = new LinkedList<>();
    public static final int FREE = 0;
// ...
}
```

API in HTML Code generiert mit Javadoc

Class Board

java.lang.Object
    Board

```
public class Board
extends Object
```

The `Board` class stores the state of a 9x9 **Sudoku** grid. It provides functionality for Sudoku solvers. A *(Priority) Queue* can store a candidate list for each cell. It employs the class `CandidateList`, which implements the `Comparable` interface based on the length of the candidate list, providing a fast access of the cell with the smallest number of possible candidates.

A **Sudoku Solver** based on *backtracking* is provided in the class `SudokuSolver`.

Author:

Benjamin Blankertz

Field Summary

Fields

Modifier and Type	Field	Description
`protected Queue<CandidateList>`	`candidates`	
`static final int`	`FREE`	

Beispiel für Doc Kommentare (Methode)

```java
/**
 * Retrieve the value (digit) of a given cell of a sudoku grid.
 *
 * @param cell in the sudoku grid
 * @return the value (digit) of the given cell
 * @throws IndexOutOfBoundsException if {@code cell} coordinates are not in 0...8.
 */

protected int getField(Position cell) {
    if (cell.x < 0  cell.x >= Nsize  cell.y < 0  cell.y >= Nsize) {
        throw new IndexOutOfBoundsException();
    }
        return field[cell.y][cell.x];
}
```

API in HTML Code generiert mit Javadoc

Method Details

getField

```
protected int getField(Position cell)
```

Retrieve the value (digit) of a given cell of a sudoku grid.

Parameters:

`cell` - in the sudoku grid

Returns:

the value (digit) of the given cell

Throws:

`IndexOutOfBoundsException` - if `cell` coordinates are not in 0...8.

Fehler- und Ausnahmebehandlung zur Laufzeit

Methoden, um Fehler zu vermeiden oder aussagekräftige Reaktionen zu veranlassen:

- **APIs**: Übereinkunft zwischen Personen, die ADT implementieren und denen, die sie nutzen
- **Schnittstellen and abstrakte Methoden**: Fehler beim Kompilieren aufdecken
- Viele Fehler lassen sich allerdings erst zur Laufzeit feststellen.
- **Ausnahmen** (*exceptions*): Abfangen von Fehlern durch die Implementation
- **Assertionen** (*assertions*): frühzeitig ungültige Bedingungen abfangen

Programmiermodel *Design-by-Contract*

Gemäß dem Programmiermodel *Design-by-Contract* (Entwurf gemäß Vertrag) wird für jede Methode definiert:

- Vorbedingungen (*preconditions*): Bedingungen, die der Client beim Aufruf einhalten muss.
- Nachbedingungen (*postconditions*): Bedingungen, die die Implementation bezüglich der Rückgabe der Methode zusichert.
- Nebeneffekte (*side effects*): Zustandsänderungen, die die Methode verursachen kann.

Im Minimalfall kann der *Design by Contract* in der API definiert werden. Darüber hinaus sollten die Bedingungen über *exceptions* und *assertions* geprüft werden.

Der *Design by Contract* ergänzt Verfahren wie *unit testing* zur Fehlervermeidung.

Ausnahmenbehandlung

- C: Fehler werden durch speziellen Rückgabewert angezeigt, z.B. `-1`, `0` oder `NULL`. Nachteile:
  - Damit liegt die Verantwortung, Fehler zu behandeln ganz in der Verantwortung der Programmierenden.
  - Der Programmcode wird durch (teilweise kaskadierte) `if`-Abfragen des Rückgabewertes schlechter lesbar.
- In Java können Methoden bei Ausnahmen und Fehler eine **`exception`** auslösen, bzw. 'werfen' (**`throw`**).
- Die aufrufende Methode kann Exceptions mit **`try`** ... **`catch`** Anweisungen abfangen und entsprechend reagieren, oder nach oben weiterleiten.
- Exceptions die nirgends abgefangen werden führen zu einem Programmabburch.
- In den Java Bibliotheken gibt es verschiedene Exceptions. Bei `catch` gibt man als Argument an, welche Exception Art(en) abgefangen weden soll(en). Man kann auch eigene Exceptions definieren, die von der Klasse `Exception` aus `java.lang` abgeleitet werden müssen.

Beispiel: ohne Ausnahmenbehandlung

Das folgende Programm nimmt eine Eingabe in einem Dialogfenster entgegen und wandelt diese in eine `int` Zahl um.

```
public class readNumberNaiv
{
  public static void getInteger() {
    // Eingabedialog öffnen:
    String str = javax.swing.JOptionPane.showInputDialog("Zahl eingeben: ");
    int number = Integer.parseInt(str);
    System.out.println("Die Zahl " + number + " war lecker!");
  }

  public static void main(String[] args) {
    getInteger();
  }
}
```

Falls die Eingabe nicht in einen `int` umgewandelt werden kann, bricht die Methode `Integer.parseInt()` mit einer `NumberFormatException` ab.

Beispiel: Ausnahmenbehandlung

```
public class readNumber
{
  public static void getInteger()
  {
    int number = 0;
    String str = "";
    while (true) {
      try {           // Fehler im try-Block führen nicht zum Programmabbruch
                      // sondern zur Ausführung des catch-Blocks
        str = javax.swing.JOptionPane.showInputDialog("Zahl eingeben: ");
        number = Integer.parseInt(str);
        break;                              // while Schleife verlassen
      } catch (NumberFormatException e) {   // Fehlerbehandlung
        System.err.println("'" + str + "' schmeckt mir nicht.");
      }
    }
    System.out.println("Die Zahl " + number + " war lecker!");
  }

  public static void main(String[] args) { ... }   // wie zuvor
}
```

Beispiel: selbst eine Ausnahme werfen

```
public static void getInteger(int low, int high)  // Zahl zwischen low und high
{
  int number = 0;
  String str = "";
  while (true) {
    try {
      String msg = "Zahl eingeben ("  low + "-" + high + "): ";
      str = javax.swing.JOptionPane.showInputDialog(msg);
      number = Integer.parseInt(str);
      System.out.println("n = " + number);
      break;
    } catch (NumberFormatException e) {
      System.err.println("'" + str + "' schmeckt mir nicht.");
    }
  }
  if (number < low || number > high) {  // Exception auslösen
    throw new InputMismatchException("Zahl nicht im angegebenen Intervall!");
  }
  System.out.println("Die Zahl " + number + " war lecker!");
}
```

Mehr Details zu Ausnahmen gibt es unter dem unten angebenen Link.

Modultests und testgetriebene Entwicklung

- **Modultests** (*unit tests*) sind ein wichtiges Werkzeug der Softwareentwicklung.
- Sie prüfen einzelne Komponenten auf korrekte Funktionalität.
- Besonders wichtig sind sie in größeren Projekten, um sicherzustellen, dass korrekt implementierte Module auch bei Weiterentwicklungen und Überarbeitungen korrekt bleiben.
- Die nächste Teststufe auf hörerer Ebene heißt *Integrationstest* und wird hier nicht behandelt.
- Bei der **testgetriebenen Entwicklung** (*test-driven development*) werden die Tests zuerst geschrieben, also vor der zu testenden Methode.
- Dadurch wird der Anforderungsrahmen an die Implementation gesteckt.
- Dann werden die Tests bei der Entwicklung automatisiert ausgeführt.

JUnit Tests

- Die Erfahrung aus der Softwareentwicklung zeigt: Mit automatisch ausgeführten Test sinken die Fehlerraten deutlich und der Entwicklungsprozess ist schneller.
- Der Qualitätssicherung durch Tests kann natürlich nur so gut sein, wie die Tests. Diese sind also mit Bedacht zu entwerfen.
- Für Java sind **JUnit Tests** als Framework für Modultests verbreitet.
- IDEs unterstützen die Entwicklung von Modultests.
- Sie können auch im Code anzeigen, welche Teile von Tests abgedeckt sind (*coverage*).
- Siehe Demo Videos zu JUnit Tests auf ISIS.

Die Schnittstellen `Comparable` und `Comparator`

- Neben `Iterable` und `Iterator` gibt es ein weiteres Paar wichtiger Schnittstellen: **`Comparable`** und **`Comparator`**
- Diese sind allerdings kein zusammengehöriges Paar, sondern zwei Varianten für unterschiedliche Fälle.
- Klassen sollten eine dieser Schnittstellen implementieren, wenn Methoden benutzt werden sollen, die auf einer Ordnung basieren, z.B. Sortieren.
- Die Schnittstelle `Comparable` befindet sich in dem Paket `java.lang` und `Comparator` in `java.util`.

Die Schnittstellen `Comparable` und `Comparator`

Die Schnittstelle Comparable

public interface Comparable<T>

`int`	`compareTo(T o)`	vergleicht dieses Objekt mit Objekt o bezüglich einer Ordnung

- Die Schnittstelle `Comparable` sollte implementiert werden, wenn es nur eine sinnvolle Ordnung auf den Objekten der Klasse gibt (genannt 'natürliche Ordnung').

Die Schnittstelle Comparator

public interface Comparator<T>

`int`	`compare(T o1, T o2)`	vergleicht die gegeben Objekte bezüglich einer Ordnung
	`...`	weitere Methoden, Implementation optional

- Wenn es alternative Möglichkeiten gibt, kann die Klasse mehrere Ordnungen über die `Comparator` Schnittstelle definieren.
- So kann z.B. eine Sortierfunktion mit unterschiedlichen Ordnungen aufgerufen werden.

Die Schnittstellen `Comparable` und `Comparator`

`v.compareTo(w)` bzw. `compare(v, w)` sollen Werte `-1`, `0`, oder `1` zurückliefern:

- `-1` für v<w,
- `0` für v=w und
- `1` für v>w.

wobei die rechte Seite eine sinnvolle Ordnung für die Elemente der Klasse darstellt:

- Die Relation muss für alle Paare von Objekten definiert sein (total)
- Für alle v gilt v=v, d.h. `v.compareTo(v)==0` (reflexiv)
- Wenn v<w ist, dann auch w>v; wenn v=w dann auch w=v (anti/symmetrisch)
- Aus u<v und v<w folgt u<w (transitiv)

Implementationsbeispiel Comparator 1/3

```java
import java.util.ArrayList;

public class Person {
  protected String name;
  protected int age;
  protected double height;

  public Person(String name, int age, double height) {
    this.name = name;
    this.age = age;
    this.height = height;
  }

  public String toString() {
    return "(" + name + ", " + age + "y, " + height + "cm)";
  }

  // main() Methode folgt
}
```

Implementationsbeispiel Comparator 2/3

```
import java.util.Comparator;
// Die Comparator könnten auch als anonyme Klassen im Aufruf implementiert werden.
// siehe Beispiele im git in Material/Code/Lecture03

public class SortByName implements Comparator<Person> {
  public int compare(Person person1, Person person2) {
    return person1.name.compareTo(person2.name);
  }
}

public class SortByAge implements Comparator<Person> {
  public int compare(Person person1, Person person2) {
    return Integer.compare(person1.age, person2.age);
  }
}

public class SortByHeight implements Comparator<Person> {
  public int compare(Person person1, Person person2) {
    return Double.compare(person1.height, person2.height);
  }
}
```

Implementationsbeispiel Comparator 3/3

```
// main() Methode der Klasse 'Person'

  public static void main(String[] args) {
    ArrayList<Person> personen = new ArrayList<>();
    personen.add(new Person("Peter", 80, 175.8));
    personen.add(new Person("Paul",  81, 191.7));
    personen.add(new Person("Mary",  82, 177.2));

    personen.sort(new SortByAge());
    System.out.println("Sorted by Age:\n" + personen);

    personen.sort(new SortByName());
    System.out.println("Sorted by Name:\n" + personen);

    personen.sort(new SortByHeight());
    System.out.println("Sorted by Height:\n" + personen);

    personen.sort(new SortByHeight().reversed());
    System.out.println("Descending by Height:\n" + personen);
  }
```

Implementationsbeispiel Comparable (1)

```
public class Person implements Comparable<Person> {
    protected String name;
    protected int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String toString() {
        return "(" + name + ", " + age + "y)";
    }

    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age);
    }

    // main() siehe nächste Seite
}
```

Implementationsbeispiel Comparable (2)

```
public static void main(String[] args) {
    ArrayList<Person> personen = new ArrayList<>();
    personen.add(new Person("Mary", 82));
    personen.add(new Person("Peter",80));
    personen.add(new Person("Paul", 81));

    personen.sort(null);                              // null: nutze compareTo()
    System.out.println("Sorted by Age:\n" + personen);

    Collections.sort(personen);                       // Alternative
    System.out.println("Sorted by Age:\n" + personen);

    Collections.sort(personen, Collections.reverseOrder());
    System.out.println("Descending by Age:\n" + personen);
}
```

Vorrangwarteschlange (Wdh. von IntroProg)

- Aus IntroProg: Datenstruktur Vorrangwarteschlange (*priority queue*, PQ) mit
- effizienter Implementierung durch binären Halden (*binary heaps*).
- Wie Warteschlange: Werte werden elementweise zugefügt.
- Der Abruf bei PQ nicht chronologisch, sondern nach gegebener Ordnung,
- z.B. immer das 'größte Element'
- Abrufreihenfolge (die Priorität) wird über einen `Comparator` bzw. ein `Comparable` realisiert.
- Diese Datenstruktur wird in späteren Algorithmen (insbesondere Graphalgorithmen) benötigt.

API für eine Vorrangwarteschlange

API für eine Vorrangwarteschlange

public class MaxPQ<K extends Comparable<K>>

	`public MaxPQ(int capacity)`	Erzeugt Vorrangwarteschlange mit Kapazität `capacity`.
void	`add(K item)`	Fügt ein Element hinzu.
`K`	`poll()`	Entfernt den größten Schlüssel und gibt ihn zurück.
boolean	`isEmpty()`	Prüft, ob die Warteschlange leer ist.
int	`size()`	Gibt Anzahl der Elemente zurück.

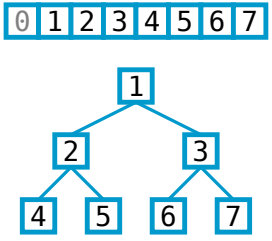
- Ebenso `MinPQ`

Eine Vorrangwarteschlange implementieren

- PQ als einfache Warteschlange, wobei `poll()` das größte Element in der unsortierten Schlange suchen müsste: Laufzeit in $\boldsymbol{O}(N)$.
- Mit einem geordneten Feld kann das größte Element zwar in $\boldsymbol{O}(1)$ gefunden werden, aber `add()` hat eine *worst case* Laufzeit in $\boldsymbol{O}(N)$.
- Mit einem binären Heap kann eine Vorrangwarteschlange mit einer Laufzeit in $\boldsymbol{O}(\log N)$ für `poll()` und `add()` realisiert werden.

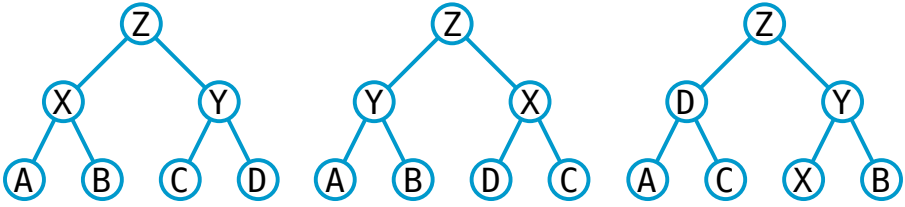
Binärer Heap

- Ein binärer Heap (Halde) ist ein Feld `A`, das als vollständiger (linksvoller) Binärbaum interpretiert wird und für den eine Ordnungsbedingung gilt.
- Die Wurzel ist `A[1]`. (Index `0` wird nicht benutzt.)
- Für Knoten k ist $\lfloor k/2 \rfloor$ der übergeordnete Knoten und $2k$ und $2k+1$ die beiden untergeordneten Knoten.



Binärer Heap

- Ein binärer Heap (Halde) ist ein Feld `A`, das als vollständiger (linksvoller) Binärbaum interpretiert wird und für den eine Ordnungsbedingung gilt.
- Die Wurzel ist `A[1]`. (Index `0` wird nicht benutzt.)
- Für Knoten k ist $\lfloor k/2 \rfloor$ der übergeordnete Knoten und $2k$ und $2k+1$ die beiden untergeordneten Knoten.
- Die Heap Ordnung besagt, dass jeder Schlüssel größer (bzw. kleiner für `MinPQ`) ist als die Schlüssel an den beiden untergeordneten Knoten.
- Dadurch sind die Positionen der Schlüssel nicht vollständig festgelegt.



Aufrechterhaltung der Heap-Ordnung (hier für MaxPQ)

- Ist *ein* Schlüssel im Heap am falschen Platz, z.B. durch Einfügen oder Ändern, so kann die Heap Ordnung durch die beiden folgenden Operationen wiederhergestellt werden.

- **`swim()`**: Ist der Schlüssel größer als der Schlüssel des übergeordneten Knoten, wird er solange nach oben getauscht, bis er an der richtigen Stelle ist ("nach oben schwimmen").

- **`sink()`**: Ist der Schlüssel kleiner als ein (oder beide) Schlüssel der untergeordneten Knoten, so wird er mit dem größeren jener beiden vertauscht - iterativ bis er richtig platziert ist ("nach unten absinken").

- Mit diesen beiden Verfahren kann die Heap Ordnung wiederhergestellt werden, siehe Vorlesung *IntroProg* oder Referenz unten.

Implementation mit Binärem Heap und Laufzeitbetrachtung

- Zum Einfügen eines Elementes fügt man es am Ende des Heaps hinzu und lässt es nach oben schwimmen (Methode `swim()`).
- Das größte Element befindet sich immer an der Wurzel des Heaps (keine Suche notwendig).
- Um es zu Entfernen, holt man das letzte Element des Heaps an die Wurzel und lässt es absinken (Methode `sink()`).
- Die Laufzeit $\boldsymbol{O}(\log N)$ ergibt sich aus der Beobachtung, dass bei `swim()` und `sink()` die Anzahl der Iterationen durch die Höhe des Binärbaums begrenzt ist. Die Höhe eines Binärbaums mit N Elementen ist $\log(N)$.

Implementation einer Vorrangwarteschlange

```
public class MaxPQ<E extends Comparable<E>>
{
  private E[] pq;
  private int N;

  public MaxPQ(int capacity)
  {
    pq = (E[]) new Comparable[capacity+1];
  }

  public int size()        { return N; }
  public boolean isEmpty() { return N == 0; }
```

Implementation einer Vorrangwarteschlange (2)

```
public void add(E e)
{
  pq[++N] = e;
  swim(N);
}

public E poll()
{
  E head = pq[1];
  swap(1, N);
  pq[N--] = null;
  sink(1);
  return head;
}

private boolean order(int i, int j) {
  return pq[i].compareTo(pq[j]) >= 0;
}
```

Implementation einer Vorrangwarteschlange (3)

```
private void swap(int i, int j) {
  E e = pq[i];  pq[i] = pq[j];  pq[j] = e;
}

private void swim(int k) {
  while (k > 1 && !order(k/2, k)) {
    swap(k/2, k);
    k = k/2;
  }
}

private void sink(int k) {
  while (2*k <= N) {
    int j = 2*k;
    if (j < N && !order(j, j+1))
      j++;
    if (order(k, j)) break;
    swap(k, j);
    k = j;
  }
}
```

Anwendungsbeispiel Vorrangwarteschlange

Die Vorrangwarteschlange bietet eine effiziente Möglichkeit für folgende Aufgabe:

- Aus einem großen Eingabestrom sollen die M kleinsten Werte herausgesucht werden.
- **Möglichkeit 1:** Unsortierte Liste mit den aktuell M kleinsten Werten speichern und neue Elemente mit all diesen Werten vergleichen: uneffizient.
- **Möglichkeit 2:** Mit einer sortierten Liste geht es schneller, bedeutet aber größeren Aufwand, um sie sortiert zu halten.
- **Möglichkeit 3:** Implementation durch eine Vorrangwarteschlange erlaubt einen Mittelweg mit einer teilweisen Sortierung, die die benötigte Funktionalität bei großer Effizienz gewährleistet.

Anwendung einer Vorrangwarteschlangen

Das folgende Client Programm extrahiert aus dem Eingabefeld `testArray` die $M = 4$ kleinsten Zahlen unter Benutzung einer `MaxPQ`:

```
public static void main(String[] args)
{
  int[] testArray = {4, 2, -17, 5, 23, 45, 0, 34, -7, 2, 0, 34};
  int M = 4;

  MaxPQ<Integer> pq = new MaxPQ<>(M+1);
  for (int k : testArray) {
    pq.add(k);
    if (pq.size() > M)
      pq.poll();
  }
  // Ausgabe der extrahierten Zahlen:
  while (pq.size() > 0) {
    int k = pq.poll();
    System.out.println("Element: " + k);
  }
}
```

Laufzeit der **`PriorityQueue`** aus den *Java Collections*

Bei der `PriorityQueue` ist zu beachten, dass ein Ändern der Priorität durch Entfernen (`remove(Object o)`) und wieder Einfügen (`add(E e)`) realisiert werden muss, was in einer linearen Laufzeit resultiert. Im Folgenden wird die `IndexPriorityQueue` eingeführt, die dies in logarithmischer Zeit erlaubt, sofern Indizes für die Elemente verfügbar sind.

PriorityQueue (*worst case*)	
`add(E e)`	$\boldsymbol{O}(\log N)$
`contains(Object o)`	$\boldsymbol{O}(N)$
`peek()`	$\boldsymbol{O}(1)$
`poll()`	$\boldsymbol{O}(\log N)$
`remove()`	$\boldsymbol{O}(\log N)$
`remove(Object o)`	$\boldsymbol{O}(N)$

Index-basierte Vorrangwarteschlangen

- Nachteil einer PQ: Löschen eines Eintrags nicht effizient: $\boldsymbol{O}(N)$.
- Außerdem wäre es praktisch, über den Index auf die Schlüssel zugreifen zu können, z.B. um einen Schlüssel zu verändern.
- Diese Möglichkeit bieten **Indizierte Vorrangwarteschlangen**.
- Über den Index können Elemente in $\boldsymbol{O}(\log N)$ gelöscht werden.

API für indizierte Vorrangwarteschlangen

API für eine indizierte Vorrangwarteschlange

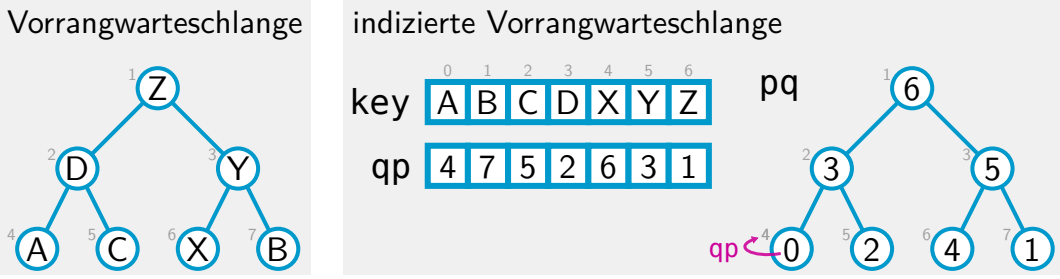
public class IndexMaxPQ<K extends Comparable<K>>

	`public IndexMaxPQ(int c)`	Erzeugt indizierte Vorrangwarteschlange mit Kapazität `c`.
void	`add(int i, K key)`	Fügt Schlüssel `key` mit Index `i` hinzu.
void	`change(int i, K key)`	Ändert den Schlüssel mit Index `i` in `key`.
void	`remove(int i)`	Löscht den Eintrag mit Index `i`.
int	`poll()`	Entfernt den Schlüssel vom Kopf und liefert seinen Index.
K	`peek()`	Liefert den Schlüssel vom Kopf der Schlange.
K	`peekIndex()`	Liefert den Index des Schlüssels vom Kopf der Schlange.
K	`keyOf(int i)`	Gibt den Schlüssel zu Index `i` zurück.
boolean	`contains(int i)`	Gibt es einen Schlüssel mit Index `i`?
boolean	`isEmpty()`	Prüft, ob die Warteschlange leer ist.
int	`size()`	Gibt Anzahl der Elemente zurück.

Durch die Index Funktionen `add`, `change` und `keyOf` beinhaltet eine indizierte Vorrangwarteschlange u.a. die Funktionalität eines Arrays.

Implementieren der indiziereten Vorrangwarteschlange

- Vorrangwarteschlangen verwenden ein Feld als binären Heap für die Schlüssel.
- Indizierte Vorrangwarteschlangen verwenden drei Felder:
- Die Schlüssel werden unter ihrem angegeben Index in einem Feld key gespeichert.
- In einem binären Heap pq werden die Schlüssel-Indizes in der Heap Ordnung bezüglich der Schlüssel gespeichert.
- Ein weiteres Feld qp speichert die inverse Abbildung, vom Schlüssel-Index zum Index im Heap.



Implementation einer indizierten Vorrangwarteschlange

```
public class IndexPQ<K extends Comparable<K>>
{
  private K[] keys;
  private int[] pq;
  private int[] qp;         // pq[qp[i]] = qp[pq[i]] = i
  private int N;
  private int sign;         // +1 IndexMaxPQ, -1 IndexMinPQ

  public IndexPQ(int capacity, int sign) {
    keys = (K[]) new Comparable[capacity];
    pq = new int[capacity + 1];
    qp = new int[capacity];
    for (int i = 0; i < capacity; i++)
      qp[i] = -1;           // qp[i]==-1 <=> es gibt keinen Schlüssel zu Index i
    this.sign = sign;
  }

  public int size()              { return N; }
  public boolean isEmpty()       { return N == 0; }
  public boolean contains(int i) { return qp[i] != -1; }
```

Implementation einer indizierten Vorrangwarteschlange (2)

```
public void add(int i, K key) {
  keys[i] = key;
  qp[i] = ++N;
  pq[N] = i;
  swim(N);
}

public void change(int i, K key) {
  keys[i] = key;
  swim(qp[i]);
  sink(qp[i]);
}

public int peekIndex()
    { return pq[1]; }
public K peek() {
    { return keys[pq[1]]; }
public K keyOf(int i)
    { return keys[i]; }
```

```
public int poll() {
  int head = pq[1];
  swap(1, N--);
  sink(1);
  qp[head] = -1;
  keys[head] = null;
  pq[N+1] = -1;
  return head;
}

public void remove(int i) {
  int index = qp[i];
  swap(index, N--);
  swim(index);
  sink(index);
  keys[i] = null;
  qp[i] = -1;
}
```

Implementation einer indizierten Vorrangwarteschlange (3)

```
// Methoden swim und sink wie zuvor
// siehe rechte Seite
// Bei order und swap kleine Änderungen

  private boolean order(int i, int j)
  {
    return sign *
      keys[pq[i]].compareTo(keys[pq[j]])
      >= 0;
  }

  private void swap(int i, int j)
  {
    int tmp = pq[i];
    pq[i] = pq[j];
    pq[j] = tmp;
    qp[pq[i]] = i;
    qp[pq[j]] = j;
  }
```

```
  private void swim(int k)
  {
    while (k > 1 && !order(k/2, k)) {
      swap(k/2, k);
      k = k/2;
    }
  }

  private void sink(int k)
  {
    while (2*k <= N) {
      int j = 2*k;
      if (j < N && !order(j, j+1))
        j++;
      if (order(k, j)) break;
      swap(k, j);
      k = j;
    }
  }
}
```

Laufzeitbetrachtung einer indizierten Vorrangwarteschlange

- Wie bei der Vorrangwarteschlange sind auch bei der indizierten Version `swim()` und `sink()` die einzigen Methoden, die Schleifen enthalten abgesehen von der Initialisierung von `qp` im Konstruktor.
- Diese beiden Methoden haben auch hier eine Laufzeit in $\boldsymbol{O}(\log N)$, wobei N für die Anzahl der Elemente in der Schlage steht.
- Entsprechend haben alle Methoden, die `swim()` oder `sink()` aufrufen, eine Laufzeit in $\boldsymbol{O}(\log N)$.
- Dabei ist zu beachten, dass auch der Aufruf beider Funktionen die Wachstumsordnung nicht ändert (konstanter Faktor 2).
- Somit ist die Laufzeit von `add()`, `change()`, `poll()` und `remove()` in $\boldsymbol{O}(\log N)$, während alle anderen Methoden eine Laufzeit in $\boldsymbol{O}(1)$ haben.
- Wie in der letzten Vorlesung ist dies eine Minimal-Implementierung, bei der viele wichtige Überprüfungen weggelassen wurden.

Laufzeiten der indizierten Vorrangwarteschlange

IndexPQ (*worst case*)	
`IndexPQ(int N, int sign)`	$\boldsymbol{O}(N)$
`add(int i, K key)`	$\boldsymbol{O}(\log N)$
`change(int i, K key)`	$\boldsymbol{O}(\log N)$
`contains(int i)`	$\boldsymbol{O}(1)$
`keyOf(int i)`	$\boldsymbol{O}(1)$
`peek()`	$\boldsymbol{O}(1)$
`peekIndex()`	$\boldsymbol{O}(1)$
`poll()`	$\boldsymbol{O}(\log N)$
`remove(int i)`	$\boldsymbol{O}(\log N)$

Verbesserung der Laufzeit

Folgende Maßnahmen können den konstanten Faktor in der Laufzeit verbessern:

- `swim()` und `sink()` können effizienter implementiert werden: In dem Heap wird nach oben bzw. nach unten die Zielposition gesucht und dann nur ein Tausch durchgeführt.
- Anstelle von `change()` können die Methoden `decreaseKey()` und `increaseKey()` implementiert werden, die jeweils nur `swim()` bzw. nur `sink()` aufrufen.
- Es gibt alternative Implementationen von (indizierten) Vorrangwarteschlangen, die eine bessere asymptotische Laufzeit haben, als die mit binären Heaps, z.B.:
- Fibonacci, Strict Fibonacci und Brodal, allerdings sind die Implementationen deutlich aufwändiger.

Rückschau

Nach dieser Vorlesung sollten Sie folgende Konzepte verinnerlicht haben:

- JavaDoc Kommentare
- Design-by-contract
- Exceptions abfangen und selbst werfen
- Vorrangwarteschlangen mit binären Heaps
- Indizierte Vorrangwarteschlangen

Anhang

Inhalt des Anhangs:

- Assertionen (*assertions*): S. 47

Assertionen

- Mit einer **Assertion** kann angezeigt werden, dass an der entsprechenden Stelle im Programmcode die angegebene Bedingung erfüllt sein muss.
- Als Anwendung einer Assertion soll die Methode `moveTo()` der `Token` Klasse sicherstellen, dass die Zielposition innerhalb des Spielfeldes liegt.
- Bisher kennt die `Token` Klasse allerdings die Größe des Spielfeldes gar nicht.
- Daher führen wir noch eine `Board` Klasse ein.
- Das Spielbrett (`Board`) enthält einen Stapel von Spielsteinen (`Token`) als Attribut.
- Jeder `Token` hat das `Board` als Attribut.
- So hat jeder Spielstein Information über die Brettgröße und `moveTo()` kann sicherstellen, dass der Stein nicht vom Brett gezogen wird.

Assertionen Beispiel (Vorbereitung)

```
public class Board
{
  public int sizeX, sizeY;      // wir machen es uns einfach: public!
  private Stack<Token> tokens;  // die Spielsteine auf dem Brett

  Board(int sizeX, int sizeY) {
    this.sizeX = sizeX;
    this.sizeY = sizeY;
    tokens = new Stack<>();
  }

  protected void addToken(Token token) {
    tokens.push(token);
  }

  public String toString() {
    return "Board of size " + sizeX + "x" + sizeY;
  }
}
```

Assertionen Beispiel

```
public class Token {
  private Board board;
  private int xPos, yPos;

  Token(Board board, int x, int y)
  {
    this.board= board;
    xPos = x;
    yPos = y;
  }

  protected void moveTo(int x, int y)
  {
    assert x >= 0 && x < board.sizeX : "x-Wert außerhalb des Spielbrettes";
    assert y >= 0 && y < board.sizeY : "y-Wert außerhalb des Spielbrettes";
    xPos= x;
    yPos= y;
  }
}
```

Assertionen Beispiel Ausführen

Um das Beispiel ausführen zu können, benötigen wir eine `main()` Methode in der `Board` Klasse:

```
  public static void main(String[] args)     // Methode in der Klasse Board
  {
    Board board = new Board(5, 7);           // Spielbrett der Größe 5x7 erzeugen
    Token token1 = new Token(board, 0, 0);   // Ein Spielstein erzeugen
    board.addToken(token1);                  //   und hinzufügen
    token1.moveTo(8, 8);                     // Auf verbotene Position setzen
    System.out.println("Token 1: " + token1);
  }
```

In der Grundeinstellung sind Assertionen bei der Ausführung ausgeschaltet. Mit der Option `-ea` bzw. `-enableassertions` werden sie in der JVM eingeschaltet. In IDEA gibt man dies bei *VM Options* unter *Run* | *Edit Configurations ...* an.

```
> javac Board.java
> java -ea Board
Exception in thread "main" java.lang.AssertionError: x-Wert außerhalb des ...
        at Token.moveTo(Token.java:18)
        at Board.main(Board.java:28)
```

Bemerkungen zu Assertionen

- Assertationen sind dazu geeignet, wichtige Aspekte des *Design-by-Contract* umzusetzen (siehe Seite 8).
- So lassen sich Vor- und Nachbedingungen sicherstellen.
- Darüber hinaus können eigene Annahmen über den Zustand in einer bestimmen Codezeile geprüft werden:

```
if (i % 3 == 0) {
  // ...
} else if (i % 3 == 1) {
  // ...
} else {
  assert i % 3 == 2 : i;
  // ...
}
```

In diesem Beispiel scheint die `assert` Bedingung sicher erfüllt zu sein (stimmt aber für `i<0` nicht). Besonders bei komplexeren Fällen kann ein `assert` hilfreich sein. Der Code im letzten `else`-Block könnte bei Änderungen in den oberen `if`-Bedingungen ungültig werden.

Weitere Informationen, auch darüber wozu Assertionen nicht benutzt werden sollen, stehen auf der unten angegebenen Internetseite.

Literatur

- Sedgewick R & Wayne K, **Introduction to Programming in Java: An Interdisciplinary Approach.** 2. Auflage, Addison-Wesley Professional, 2017.
  Onlinefassung: `https://introcs.cs.princeton.edu/java`
- Ullenboom C, **Java ist auch eine Insel.** 13. Auflage, Rheinwerk Computing, 2018.
  Onlinefassung: `http://openbook.rheinwerk-verlag.de/javainsel`
- `https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html`

Danksagung. Die Folien wurden mit LaTeX erstellt unter Verwendung vieler Pakete, u.a. beamer, listings, lstbackground, pgffor und colortbl sowie eine Vielzahl von Tipps auf `tex.stackexchange.com` und anderen Internetseiten.

Index

API
  indizierte Vorrangwarteschlange, 37
  MaxPQ, *see* API, Vorrangwarteschlange
  MinPQ, *see* API, Vorrangwarteschlange
  Priority Queue, *see* API, Vorrangwarteschlange
  Vorrangwarteschlange, 25
Assertion, 47
Assertionen, 8
assertions, 8
Ausnahmen, 8
Ausnahmenbehandlung, 12

`catch`, 10
`Comparable`, 16
`Comparator`, 16
coverage, 15

Design-by-Contract, 9

`exception`, 10, 12
exceptions, 8

Heap Ordnung, 27

`IndexPQ`
  Laufzeit, 43
Indizierte Vorrangwarteschlangen, 36

Javadoc, 3
JUnit Tests, 15

Kommentare, 2

Laufzeit
  indizierte Vorrangwarteschlange, 42
  Vorrangwarteschlange, 29

Modultests, 14

postconditions, 9
preconditions, 9
Priority Queue
  Laufzeit, 26
priority queue, 24
`PriorityQueue`, 35

side effects, 9
`sink()`, 28
`swim()`, 28

test-driven development, 14
testgetriebene Entwicklung, 14
`throw`, 10
`try`, 10

unit tests, 14

Vorrangwarteschlange, 24
  indizierte, 36
  Laufzeit, 26, 29