Inhaltsverzeichnis

Objekte mit Schlüsseln sammeln

1. Einleitung

Eine *Sammlung* ist ein Objekt, in dem man Objekte *sammeln* (d. h. hineintun, suchen und wieder entfernen) kann. Alle Sammlungsklassen in der Java-Standardbibliothek implementieren die Schnittstelle `Collection`, so dass man eine Sammlung formal auch als ein `Collection`-Objekt definieren kann.

Um Sammlungen besser zu verstehen, ist es günstig, selbst ein paar Sammlungsklassen zu programmieren. Hier sollen Sammlungen slizziert werden, in denen man Objekte sammeln kann, die einen *Schlüssel* besitzen. Wenn man eine Sammlung und einen Schlüssel hat, soll man in der Sammlung nach einem Objekt mit diesem Schlüssel suchen können.

Beispiel: Informationen über die Kunden einer Firma kann man in Objekten speichern, von denen jedes eine eindeutige Kunden-Nummer als Schlüssel hat. Informationen zu Artikeln kann man entsprechend in Objekten speichern, die eine eindeutige Artikel-Nummer als Schlüssel haben.

Sammlungen kann man auf verschiedene Weise „strukturieren“, z. B. wie eine *Reihung*, oder als *verkettete Liste*, oder als *Baum* oder als *Hash-Tabelle* (diese Strukturen werden weiter unten genauer erläutert). Einige dieser Strukturen setzen voraus, dass man die gesammelten Objekte *sortieren* kann. Wir wollen also grundsätzlich voraussetzen, dass jedes zu sammelnde Objekt einen *Schlüssel* hat, und dass man von zwei beliebigen Schlüsseln `s1` und `s2` feststellen kann, ob `s1` kleiner, gleich oder größer `s2` ist. Diese beiden Voraussetzungen werden durch die folgende Schnittstelle `SammelbarG` (`G` wie generisch) ausgedrückt:

```
1   interface SammelbarG<S extends Comparable<? super S>> {
2      S getSlue(); // Liefert den Schluessel dieses Objekts
```

Um als sammelbar (im Sinne dieser Schnittstelle `SammelbarG`) zu gelten, muss ein Objekt also einen Schlüssel eines Typs `S` und eine Methode `getSlue` besitzen, die diesen Schlüssel liefert. Dabei müssen Schlüssel (Objekte des Typs `S`) mit Objekten eines Obertyps von `S` vergleichbar sein (mit der Methode `compareTo` der Schnittstelle `Comparable`). Zur Erinnerung: Jeder Typ gilt als Obertyp von sich selbst.

Beispiel-01: Eine SammelbarG-Klasse namens KompoG

```
3   class KompoG implements SammelbarG<String> {
4      private String schluessel;
5      private String daten;
6      // ---------------------------------------------------------------------
7      KompoG(String schluessel, String daten) {
8         this.schluessel = schluessel;
9         this.daten      = daten;
10     } // Konstruktor KompoG
11
12     static int lfdNr = 0;
13
14     KompoG() {
15        lfdNr++;
16        schluessel = "Schluessel "        + lfdNr;
17        daten      = "Daten Daten Daten " + lfdNr;
18     } // Standard-Konstruktor KompoG
19     // ---------------------------------------------------------------------
20     public String getSlue() {return schluessel;}
21     // ---------------------------------------------------------------------
22     public String toString() {
23        return "KompoG, Slue: " + schluessel + ", Daten: " + daten;
24     } // toString
25     // ---------------------------------------------------------------------
26     // Eine "vernuenftige" equals-Methode ist noetig, damit beim Testen die
27     // Gleichheit zweier KompoG-Objekte (die nicht identisch sind) richtig
28     // erkannt werden kann:
29     public boolean equals(KompoG k) {
30        return this.schluessel.equals(k.schluessel) &&
31               this.daten     .equals(k.daten);
32     } // equals
33     // ---------------------------------------------------------------------
34  } // class KompoG
```

Die Klasse String implementiert die Schnittstelle Comparable<String> (siehe Dokumentation der Standardbibliothek). Somit sind String-Objekte als Schlüssel geeignet (oder: Der Typ String ist als Parameter der Schnittstelle SammelbarG geeignet).

Eine *Sammlung* (von Objekten mit Schlüsseln) ist vor allem durch 2 „zusammenhängende Typen“ gekennzeichnet: Durch den Typ K der Komponenten, die man in der Sammlung sammeln kann und den Typ S der Schlüssel dieser Komponenten. Die folgende Schnittstelle SammlungG (G wie generisch) ist eine Art „kleine, aber selbstgeschnitzte Alternative“ zur Standardschnittstelle Collection:

```
35  interface SammlungG
36     <  S extends Comparable<? super S>, //  Schluessel-Typ
37        K extends SammelbarG<S>          // Komponenten-Typ
38     >
39  {
40     public boolean fuegeEin(K kompoNeu);
41     public K       suche   (S schluessel);
42     public boolean entferne(S schluessel);
43     public void    print   ();
44     public int     size    ();
45  } // interface SammlungG
```

Eine kommentierte Version dieser Schnittstelle findet man in der Datei SammelbarG.java bei den Beispielprogrammen.

In Zeile 37 wird festgelegt, dass die Kommponenten unserer Sammlungen sammelbar (im Sinne der Schnittstelle SammelbarG<S>) sein sollen. Die Zeile 36 drückt aus, dass der Schlüsseltyp S als Parameter der Schnittstelle SammelbarG (siehe oben) geeignet sein muss.

Aufrufe der Methoden `fuegeEin` und `entferne` können *misslingen* (z. B. weil die Sammlung voll ist bzw. kein Objekt mit dem angegebenen Schlüssel vorhanden ist). In solchen Fällen liefern die Methoden als Ergebnis `false`. *Gelingt* ein Aufruf liefern sie `true`.

Ab jetzt setzen wir also voraus, dass alle unsere (selbstprogrammierten) Sammlungsklassen die Schnittstelle `SammlungG<S, K>` implementieren.

Im Folgenden sollen unterschiedlich strukturierte Sammlungen dargestellt und diskutiert werden. Vor allem wollen wir untersuchen, wie schnell (oder langsam) die Operationen *einfügen, suchen* und *entfernen* sind. Allgemein wird das natürlich mindestens von folgenden Faktoren abhängen:

1. Von der *Struktur* der Sammlung (unsortierte Reihung, sortierte Reihung, ... etc.)
2. Von der *Anzahl* der Komponenten, die sich bereits in der Sammlung befinden
3. Davon, ob es sich um einen *günstigsten*, einen *durchschnittlichen* oder einen *ungünstigsten* Fall handelt.

Der letzte Punkt soll anhand eines Beispiels erläutert werden: Wenn wir in einer *unsortierten Reihung* nach einem Objekt (mit einem bestimmten Schluessel) suchen, indem wir "von links nach rechts durch die Reihung gehen", dann finden wir das Objekt im *günstigsten* Fall als *erstes*, in einem *durchschnittlichen* Fall etwa in der *Mitte* der Reihung und im *ungünstigsten* Fall am *Ende* der Reihung.

Speziell beim *Suchen* ist es sinnvoll noch *zwei weitere Fälle* zu unterscheiden: Den *positiven* Fall, in dem wir das gesuchte Objekt tatsächlich *finden* und den *negativen* Fall, in dem das Objekt nicht in der Sammlung vorhanden ist und wir es deshalb *nicht finden*.

Bei einer *unsortierten Reihung* gibt es beim *Suchen im negativen Fall* keinen Unterschied zwischen einem *günstigsten*, einem *durchschnittlichen* und einem *ungünstigsten* Fall (wir müssen die Reihung immer *vollständig durchsuchen* um sicher zu sein, dass das gesuchte Objekt *nicht* vorhanden ist). Bei anderen Arten von Sammlungen unterscheiden sich aber auch beim *Suchen im negativen Fall* günstigste, durchschnittliche und ungünstigste Fälle voneinander.

Den *Zeitbedarf* einer Operation (z.B. der Operation *Suchen*) drücken wir im folgenden *nicht* in *Sekunden* oder *Stunden*, sondern in *Schritten* aus. Dabei ist *ein Schritt* irgendeine Folge von Befehlen, die der Ausführer innerhalb einer beliebigen, aber *festen Zeit* ausführen kann. Insbesondere darf die Zeit für die Ausführung eines Schritts *nicht* von der momentanen *Größe der Sammlung* (d.h. von der *Anzahl der Komponenten* in der Sammlung) abhängen.

Beispiel-02: Ein Vergleich zweier `int`-Werte oder eine Addition zweier `long`-Werte kann als *ein Schritt* gezählt werden.

Beispiel-03: Ein Vergleich von 500 Paaren von `int`-Werten gefolgt von einer Addition von 2000 `long`-Werten kann als *ein Schritt* gezählt werden.

Beispiel-04: Eine Addition zweier `BigInteger`-Objekte kann nur dann als *ein Schritt* gezählt werden, wenn fest steht, dass die Objekte eine bestimmte feste Größe (z.B. 50 Dezimalziffern oder 500 Binärziffern etc.) nicht überschreiten. Sonst kann eine solche Addition *nicht* als ein Schritt gezählt werden.

Beispiel-05: Ein Vergleich zweier `String`-Objekte kann nur dann als *ein Schritt* gezählt werden, wenn feststeht, dass die Objekte eine bestimmte feste Größe (z.B. 50 Zeichen oder 500 Zeichen etc.) nicht überschreiten. Sonst kann ein solcher Vergleich *nicht* als ein Schritt gezählt werden.

Die *Größe der Sammlung* (d.h. die *Anzahl der Komponenten*, die sich bereits darin befinden) werden wir meist mit `N` bezeichnen und die Anzahl der Schritte, die für eine bestimmte Operation erforderlich sind, mit `S`. Besonders interessant ist dann die Frage: Wie hängt `S` von `N` ab? Wie verändert sich `S`, wenn wir `N` um einen Faktor `2` oder `3` oder ... vergrößern? Wächst `S` dann auch um den gleichen Faktor `2` bzw. `3` bzw ... ? Oder wächst `S` dann quadratisch um den Faktor `4` bzw. `9` bzw. ... ? Oder wächst

S dann mit der dritten Potenz um den Faktor 8, 27, ... ? Oder wächst S langsamer als N? Oder wächst S überhaupt nicht?

2. Unsortierte Reihungen

Aufgabe: Schreiben Sie eine Klasse `UnsortierteReihung<S, K>`, die die Schnittstelle `SammlungG<S, K>` implementiert und die Sammlung als *unsortierte Reihung* realisiert (d.h. beim *Einfügen* werden die Komponenten "einfach so wie sie kommen" hintereinander in die Reihung eingefügt).

Sei UR ein Objekt einer `UnsortierteReihung`-Klasse, in das bereits N Komponenten eingefügt wurden.

Frage UR1: Wie viele Schritte sind nötig, um ein weiteres Objekt in die Sammlung UR *einzufügen*?
Frage UR2: Wie viele Schritte sind nötig, um ein bestimmtes Objekt (welches sich tatsächlich in UR befindet) zu *suchen* (*Suchen im positiven Fall*)? In einem günstigsten Fall? In einem ungünstigsten Fall? In einem durchschnittlichen Fall?

Frage UR3: Wie viele Schritte sind nötig um festzustellen, dass sich ein bestimmtes Objekt (ein Objekt mit einem bestimmten Schlüssel) *nicht* in UR befindet (*Suchen im negativen Fall*)?

Frage UR4: Wie viele Schritte sind nötig um ein Objekt (mit einem bestimmten Schlüssel) zu suchen und aus der Sammlung UR zu *entfernen*?

Implementierungshinweis: In einer generischen Klasse `K<T>` darf man keine Reihung mit dem Komponententyp T vereinbaren. Eine Vereinbarung wie die folgende ist also *verboten*:

```
46  class K<T> {
47     T[] otto = new T[17]; // T als Komponententyp einer Reihung ist verboten
48     ...
```

Diese Einschränkung hängt mit dem grundlegenden Modell zusammen, nach dem generische Klassen in Java realisiert werden (und ist „ein Teil des Preises, den man für die wichtigen Vorteile dieses Modells zahlen muss“).

Man darf in einer generischen Klasse `K<T>` aber z. B. ein Objekt der Klasse `ArrayList<T>` vereinbaren und initialisieren, z. B. so:

```
49  class K<T> {
50     ArrayList<T> otto = new ArrayList<T>(17); // So darf man T benutzen
51     ...
```

Ein `ArrayList`-Objekt wird vom Ausführer im Kern als eine *Reihung* realisiert. Wenn man die Kapazität eines `ArrayList`-Objekts bei seiner Erzeugung festlegt und danach nie wieder verändert, stimmen praktisch alle wichtigen Eigenschaften des `ArrayList`-Objekts mit dem „seiner Reihung“ überein.

Wenn in diesem und den folgenden Abschnitten von (unsortierten bzw. sortierten) *Reihungen* die Rede ist, dürfen (und müssen) Sie statt dessen `ArrayList`-Objekte verwenden, deren Kapazität Sie nie verändern. Sie sollen die `ArrayList`-Objekte also nur „als Ersatz für Reihungen“ verwenden, und den zusätzlichen Komfort, den die Klasse `ArrayList` bietet, *nicht* in Anspruch nehmen. Ende des Implementierungshinweises.

3. Sortierte Reihungen

Der Algorithmus *Binäres Suchen* (binary search) ist ein besonders effizienter Algorithmus zum *Suchen* eines Objekts in einer *sortierten Reihung*. Er wird in der Vorlesung genauer besprochen.

Aufgabe: Schreiben Sie eine Klasse `SortierteReihung<S, K>`, die die Schnittstelle `SammlungG<S, K>` implementiert und die Sammlung als *sortierte Reihung* (bzw. als `ArrayList`-Objekt mit unveränderter Kapazität) realisiert (d.h. beim Einfügen werden die Komponenten entspre-

chend ihrem Schlüssel *sortiert* in die Reihung eingefügt). Das Suchen soll nach dem (schnellen) Algorithmus *Binäres Suchen* erfolgen.

Aufgabe: Wie viele Schritte sind beim *binären Suchen* maximal erforderlich, wenn die Sammlung 1000 Objekte (bzw. eine Million Objekte bzw. eine Milliarde Objekte) enthält?

Sei `SR` ein Objekt einer `SortierteReihung`-Klasse, in das bereits `N` Komponenten eingefügt wurden.

Frage SR1: Wie viele Schritte sind nötig, um ein weiteres Objekt in die Sammlung `SR` *einzufügen*? In einem *günstigsten* Fall? In einem *ungünstigsten* Fall? In einem *durchschnittlichen* Fall?

Frage SR2: Wie viele Schritte sind nötig, um ein bestimmtes Objekt (welches sich tatsächlich in `SR` befindet) zu *suchen* (*Suchen im positiven Fall*)? In einem *günstigsten* Fall? In einem *ungünstigsten* Fall? In einem *durchschnittlichen* Fall?

Frage SR3: Wie viele Schritte sind nötig um festzustellen, dass sich ein bestimmtes Objekt (ein Objekt mit einem bestimmten Schlüssel) *nicht* in `SR` befindet (*Suchen im negativen Fall*)?

Frage SR4: Wie viele Schritte sind nötig um ein Objekt (mit einem bestimmten Schlüssel) zu suchen und aus der Sammlung `SR` zu *entfernen*?

Aufgabe: Vergleichen Sie Sammlungen, die als *unsortierte* bzw. als *sortierte Reihungen* realisiert wurden. Welche Operationen (*einfügen*, *suchen*, *entfernen*) gehen bei welcher Struktur schneller? Unter welchen Umständen würden Sie die eine bzw. die andere Struktur wählen?

4. Verkettete Listen

In einer *sortierten Reihung* kann man sehr *schnell* suchen (mit dem Algorithmus *Binäres Suchen*). Eine sortierte Reihungen hat aber auch zwei Nachteile:

1. Eine Reihung kann nicht "verlängert" werden
2. Um ein Objekt *einzufügen* oder zu *entfernen* muss man im durchschnittlichen Fall *die Hälfte aller Objekte* in der Sammlung um eine Indexposition verschieben (beim *Einfügen* um eine Position nach *rechts* und beim *Entfernen* um eine Position nach *links*).

Eine (einfach) *verkettete Liste* besteht im Wesentlichen aus Knoten-Objekten, von denen jedes eine *Referenz* enthält. Diese Referenz hat entweder den Wert *null* (dann ist dies der *letzte* Knoten der Liste) oder er zeigt auf den *nächsten Knoten* der Liste.

Eine (unsortierte) *verkettete Liste* hat im Vergleich zu einer *sortierten Reihung* 3 Vorteile:

1. Man kann sie jederzeit "verlängern"
2. Das Einfügen eines Objektes ("vorn in der Liste") kostet immer nur *einen Schritt*, unabhängig von der Größe der Sammlung.
3. Wenn man ein Knoten-Objekt aus der Liste entfernt, kann sein Speicherplatz in aller Regel wiederverwendet werden (z. B. zum Erzeugen neuer Knoten-Objekte). Bei einer Reihung gibt es keinen entsprechenden Effekt.

Andererseits kann man in einer verketteten Liste *nicht* binär suchen, sondern muss sequentiell alle Objekte in der Sammlung der Reihe nach prüfen.

Aufgabe: Vergleichen Sie die Kosten der Operation *Entfernen* bei einer *sortierten Reihung* und einer *unsortierten Liste*.

Aufgabe: Schreiben Sie eine Klasse `VerketteteListe<S, K>`, die die Schnittstelle `SammlungG<S, K>` implementiert und die Sammlung als (unsortierte, einfach) *verkettete Liste* realisiert.

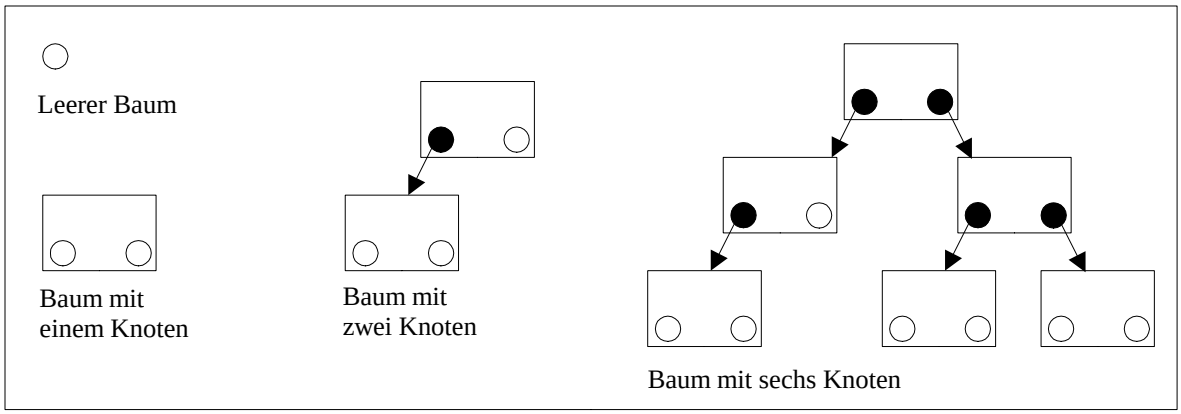
Aufgabe: Was passiert, wenn man anstelle einer *unsortierten* verketteten Liste eine *sortierte* verkettete Liste verwendet? Welche Operationen werden dadurch *schneller*, welche werden *langsamer*?

5. Binäre Bäume

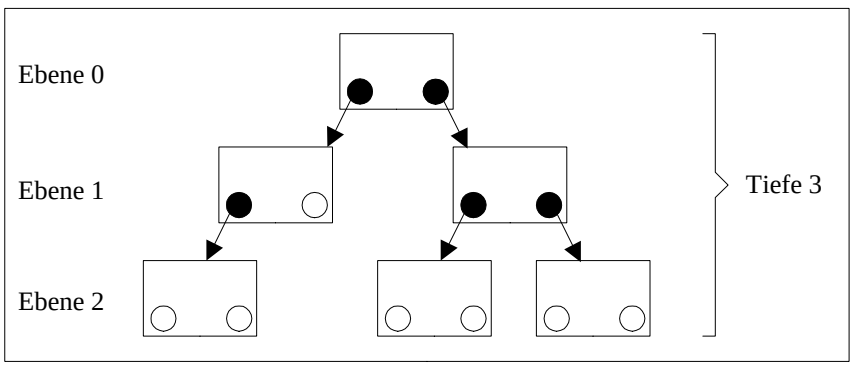
Def.: Ein *binärer Baum* ist entweder *leer* oder er besteht aus *einem Knoten*, an dem *zwei binäre Bäume* hängen.

Die beiden Bäume, die an einem Knoten hängen, bezeichnet man auch als *linken* bzw. *rechten Unterbaum* des Knotens. Die *Reihenfolge* dieser Unterbäume ist signifikant.

Man beachte, dass die obige Definition *rekursiv* ist: Der Begriff *binärer Baum* wird mit Hilfe des Begriffs *binärer Baum* definiert. Man bezeichnet Bäume deshalb häufig auch als *rekursive Datenstrukturen*. Hier als Beispiel eine graphische Darstellung von ein paar binären Bäumen:



Die Knoten eines Baumes kann man sich auf *Schichten* oder *Ebenen* angeordnet denken. Die *Anzahl* dieser Ebenen bezeichnet man auch als *Tiefe* des Baumes:



Aufgabe: *Wie viele Knoten* können (höchstens) auf der der *Ebene 0*, auf der *Ebene 1*, auf der *Ebene 2*, auf der *Ebene 3* und allgemein auf der *Ebene n* stehen?

Aufgabe: *Wie viele Knoten* kann ein binärer Baum der *Tiefe 1*, der *Tiefe 2*, der *Tiefe 3* und allgemein der *Tiefe n* (höchstens) enthalten?

Aufgabe: Mindestens *wie tief* muss ein binärer Baum sein, der *1000 Knoten* enthält? Ein Baum mit einer *Million* (d.h. 10^6) *Knoten*? Ein Baum mit einer *Milliarde* (d.h. 10^9) *Knoten*? Wie kann man allgemein aus der Anzahl *N* der Knoten die (Mindest-) *Tiefe* eines binären Baumes berechnen?

Def.: Ein Baum heisst *perfekt balanciert*, wenn alle seine Ebenen (bis auf die unterste) *maximal viele Knoten* enthalten. Die unterste Ebene darf, muss aber nicht voll besetzt sein.

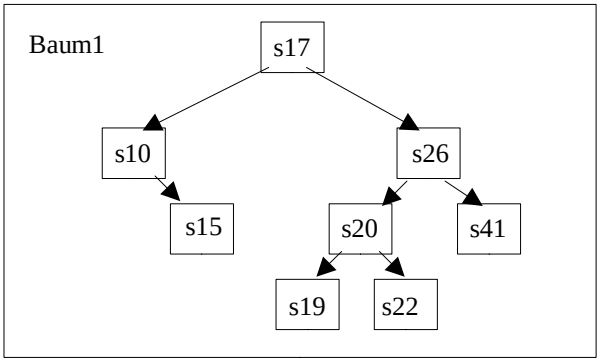
Ein *Weg* in einem Baum führt vom *Wurzelknoten* (d.h. vom "Anfang des Baumes") bis zu einem *Blattknoten* (d.h. bis zu einem "Ende des Baumes"). Unter der *Länge* eines solchen Weges versteht man die *Anzahl* der Knoten, "die auf dem Weg liegen". In einem *perfekt balancierten* Baum sind alle

Wege fast *gleich lang* (maximale Differenz: 1). Ein Baum ist *einigermaßen balanciert*, wenn kein Weg mehr als *doppelt* so lang ist wie der kürzeste Weg.

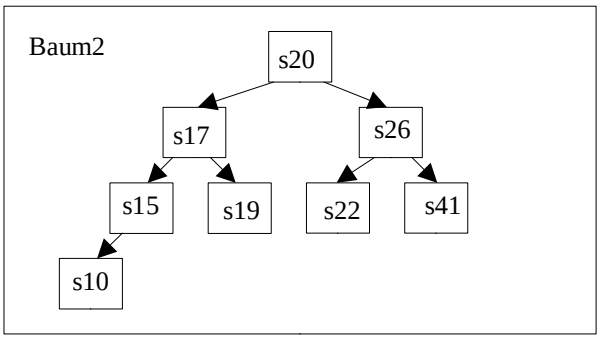
Aufgabe: Wie sieht ein *maximal unbalancierter* Baum aus? Wie viele Blätter hat ein solcher Baum? Mit welcher Datenstruktur hat ein maximal unbalancierter Baum grosse Ähnlichkeit?
Wir betrachten hier nur solche Bäume, bei denen jeder Knoten (irgendwelche Daten und) einen *Schlüssel* hat, der zu einer total geordneten Menge gehört (d.h. man kann von zwei solchen Schlüsseln nicht nur feststellen, ob sie *gleich* oder *ungleich* sind, sondern auch, ob der eine *größer* ist als der andere).

Def.: Ein binärer Baum ist *sortiert*, wenn für jeden Knoten K gilt: Der Schlüssel von K ist
größer als alle Schlüssel in seinem *linken* Unterbaum und
kleiner als alle Schlüssel in seinem *rechten* Unterbaum.

Hier ein Beispiel für einen sortierten binären Baum:



Wenn zwei *sortierte Reihungen* die gleichen Schlüssel enthalten, dann sehen sie "genau gleich" aus. Für *sortierte Bäume* gilt das *nicht*. Hier ein sortierter Baum2, der die *gleichen Schlüssel* (s10, s15, s17, s19, s20, s22, s26 und s41) enthält wie der Baum1, aber ganz *anders strukturiert* ist:



Aufgabe (mit Papier und Bleistift): Fügen Sie die folgenden Schlüssel in der angegebenen Reihenfolge in einen (anfangs leeren) binären Baum ein: s50, s20, s33, s75, s15, s80, s60 .

Aufgabe: Fügen Sie dieselben Schlüssel in folgender Reihenfolge in einen (anfangs leeren) binären Baum ein: s50, s60, s33, s20, s75, s15, s80 .

Aufgabe: Fügen Sie dieselben Schlüssel in folgender Reihenfolge in einen (anfangs leeren) binären Baum ein: s15, s20, s33, s50, s60, s75, s80 .

Aufgabe: Betrachten Sie noch einmal den oben angegebenen Baum1. In welcher Reihenfolge könnten die Schlüssel eingefügt worden sein?

Wie kann man in einem sortierten, binären Baum nach einem Knoten mit einem bestimmten Schlüssel *suchen*? Wie viele Schritte sind beim Suchen in einem *ungünstigsten Fall* nötig, wenn der Baum *per-*

fekt balanciert ist? Wenn der Baum wenigstens *einigermaßen balanciert* ist? Wenn der Baum *maximal unbalanciert* ist?

Aufgabe: Schreiben Sie eine Klasse `BinBaum<S, K>`, die die Schnittstelle `SammlungG<S, K>` implementiert und die Sammlung als (sortierten) *binären Baum* realisiert. Programmieren Sie zuerst nur die Operationen *fuegeEin* und *suche* (die Operation *entferne* ist ziemlich kompliziert). Besonders elegant und leicht verständlich kann Ihre Lösung werden, wenn Sie *Rekursion* verwenden.

Binäre Bäume kombinieren die positiven Eigenschaften von *sortierten Reihungen* und von *Listen*: Das *Suchen* in einem binären Baum geht so schnell wie das *binäre Suchen* in einer sortierten Reihung und beim *Einfügen* muss man nur "ein paar Referenzen umbiegen" (wie bei einer Liste) und nicht "viele Objekte um eine Position verschieben" (wie bei einer sortierten Reihung).

6. Hash-Tabellen

Erstaunlicherweise gibt es ein Suchverfahren, welches noch *schneller* ist als das *binäre Suchen* in einer *sortierten Reihung* oder das Suchen in einem *binären Baum*. Um es anzuwenden, muss man die Objekte in einer sogenannten *Hash-Tabelle* abspeichern. Solche Hash-Tabellen gibt es in zahlreichen Varianten. Hier soll nur ihr *Grundprinzip* anhand einer besonders einfachen Varianten dargestellt werden.

Def.: Eine *Hash-Tabelle* ist eine *Reihung* von *Listen*.

Etwas konkreter: Jede Komponente der Reihung ist der Beginn einer Liste und hat entweder den Wert *null* (dann ist diese Liste noch leer) oder enthält eine Referenz auf das erste Objekt der Liste.
Die *Reihung*, aus der eine Hash-Tabelle besteht, hat eine *feste*, unveränderbare Länge (z.B. die Länge 10 oder die Länge 10.000 oder ... , je nach Anwendung). Die einzelnen *Listen* in der Reihung haben anfänglich alle die Länge 0, können aber jederzeit *verlängert* werden, indem man weitere Objekte einfügt. Normalerweise sind die einzelnen Listen *unsortiert*.

In der Hash-Tabelle speichert man Objekte ab, die im einfachsten Fall nur aus einem *Schlüssel* (und in realistischeren Fällen aus einem *Schlüssel* und irgendwelchen *dazugehörigen Daten*) bestehen. Als konkrestes Beispiel wollen wir folgenden Fall betrachten: Die 14 Schlüssel (vom Typ `String`)

```
"Ali"  "Babsy" "Alfred" "Arno" "Alice" "Benno" "Kurt"
"Alex" "Angy"  "Bine"   "Max"  "Franz" "Susi"  "Alf"
```

sollen (ohne weitere dazugehörige Daten) in eine Hash-Tabelle der *Länge 10* eingefügt werden. Nach dem Einfügen soll es dann möglich sein, von einem beliebigen Schlüssel (String) zu prüfen, ob er in der Hash-Tabelle vorkommt oder nicht.

Das *Einfügen* eines Objektes in eine Hash-Tabelle erfolgt in *zwei Schritten*:

Schritt E1: Aus dem Schlüssel des Objekts wird mit einer sogenannten *Hashfunktion* ein *Reihungsindex* berechnet (in unserem Beispiel: eine Zahl zwischen 0 und 9).

Schritt E2: Dann wird das Objekt in die (dem Index) entsprechende Liste *eingefügt*.

Das Suchen eines Schlüssels in der Hash-Tabelle erfolgt ganz entsprechend und ebenfalls in zwei *Schritten*:

Schritt S1: Aus dem gesuchten Schlüssel wird mit der *Hashfunktion* (mit derselben wie im Schritt E1) ein *Reihungsindex* berechnet (im Beispiel: eine Zahl zwischen 0 und 9).

Schritt S2: Dann wird der Schlüssel in der (dem Index) entsprechenden Liste *gesucht*.

Das Geheimnis einer Hash-Tabelle steckt im Wesentlichen in der verwendeten *Hashfunktion*. Es gibt sehr viele verschiedene Hashfunktionen. Was sie leisten und was sie gut oder schlecht macht soll anhand einiger konkreter Beispiele erläutert werden.

Jede Hashfunktion erwartet einen *Schlüssel* als Parameter (in unserem Beispiel ist das ein Parameter vom Typ `String`) und liefert als Ergebnis einen *Index* der Hash-Tabelle (im Beispiel ein `int`-Wert zwischen 0 und 9). Beginnen wir mit einer der schlechtesten Hashfunktionen die es gibt:

```
1   int hash01(String s) {
2      return 3;
3   }
```

Mit dieser (pessimale) Hashfunktion werden alle Objekte in die Liste `3` der Hash-Tabelle eingefügt und die anderen neun Listen (`0` bis `2` und `4` bis `9`) bleiben *leer*. Hier eine Darstellung der Hash-Tabelle nach dem Einfügen aller 14 Beispiel-Schlüssel mit der Hashfunktion `hash01`:

```
Index| Elemente in den Listen
```

```
  0  |
  1  |
  2  |
  3  | Ali Babsy Alfred Arno Alice Benno Kurt Alex Angy Bine Max Franz Susi Alf
  4  |
  5  |
  6  |
  7  |
  8  |
  9  |
Suchschritte insgesamt mit hash01: 105
```

Die letzte Zeile bedeutet: Wenn man jeden Schlüssel, der in der Hash-Tabelle vorkommt, *einmal* sucht, braucht man dazu insgesamt (und "im Wesentlichen") 105 *Listen-Suchschritte*. Den Schlüssel `Ali` findet man nach *einem* Schritt, für `Babsy` braucht man *2 Schritte*, für `Alfred` *3 Schritte*, ... und für `Alf` *14 Schritte*, macht insgesamt 105 Schritte. Diese Zahl ist ein Maß für die Güte (oder: für den Mangel an Güte) der verwendeten Hashfunktion.

Mit der Hashfunktion `hash01` dauert das Einfügen eines Objekts und das Suchen nach einem Schlüssel etwa so lange wie bei einer *Liste* (hinzu kommt sogar noch je ein Aufruf der Hash-Funktion, der allerdings häufig vernachläßigt werden kann).

Aufgabe: Beschreiben Sie 9 weitere Hashfunktionen, die genauso schlecht sind wie die Funktion `hash01`.

Die folgende Funktion `hash02` ist schon etwas besser als `hash01`:

```
4   int hash02(String s) {
5      if (s.charAt(0) % 2 == 0) {
6         return 3;
7      } else {
8         return 4;
9      }
10  }
```

Je nachdem ob das erste Zeichen des Schlüssels (`s.charAt(0)`) durch eine *gerade* oder durch eine *ungerade* Zahl codiert wird, liefert diese Hashfunktion den Index `3` oder den Index `4`. Alle Objekte werden also in die Liste `3` oder in die Liste `4` eingefügt, die Listen `0` bis `2` und `5` bis `9` bleiben garantiert leer. Hier eine Darstellung der Hash-Tabelle, nachdem alle 14 Beispiel-Schlüssel mit der Hashfunktion `hash02` eingefügt wurden:

```
Index| Elemente in den Listen
  0  |
  1  |
  2  |
  3  | Babsy Benno Bine Franz
  4  | Ali Alfred Arno Alice Kurt Alex Angy Max Susi Alf
  5  |
  6  |
  7  |
  8  |
  9  |
Suchschritte insgesamt mit hash02: 65
```

An der letzten Zeile kann man erkennen, dass die Funktion `hash02` deutlich *besser* ist als `hash01`.

Hier eine *noch* bessere Hashfunktion:

```
11  int hash03(String s) {
12     return s.charAt(0) % LAENGE_DER_HASHTAB;
```

```
13  }
```

Diese Hashfunktion berechnet den Index aus dem *ersten Zeichen des Schlüssels*. Die Operation `% LAENGE_DER_HASHTAB` stellt sicher, dass wir immer einen gültigen Index der Hash-Tabelle bekommen. Hier eine Darstellung der (mit Hilfe von `hash03`) gefüllten Hash-Tabelle:

```
Index| Elemente in den Listen
  0  | Franz
  1  |
  2  |
  3  | Susi
  4  |
  5  | Ali Alfred Arno Alice Kurt Alex Angy Alf
  6  | Babsy Benno Bine
  7  | Max
  8  |
  9  |
Suchschritte insgesamt mit hash03: 45
```

Man sieht: Die Funktion `hash03` bewirkt, dass alle Schlüssel mit gleichem Anfangsbuchstaben in dieselbe Liste kommen. Allerdings können in einer Liste auch Schlüssel mit verschiedenen Anfangsbuchstaben stehen (weil z.B. `'A' % 10` gleich `'K' % 10` gleich `5` ist).

Aufgabe: Die Funktion `hash03` bewirkt, dass alle mit `'A'` und alle mit `'K'` beginnenden Schlüssel in die Liste `5` kommen. Nennen Sie einen weiteren Anfangsbuchstaben, der von `hash03` der Liste `5` zugeordnet wird.

Für ein genaueres Verständnis von Hashfunktionen besonders wichtig ist die folgende Tatsache: Ob die Funktion `hash03` besonders gut oder ziemlich schlecht oder mittelmäßig ist, kann man nicht allein anhand der Funktion selbst entscheiden. Vielmehr muss man auch die Schlüssel berücksichtigen, auf die man sie anwendet. Für die 14 Beispiel-Schlüssel ist `hash03` nicht besonders gut, denn sie läßt `5` der `10` Listen unserer Hash-Tabelle leer und bewirkt, dass 8 der 14 Schlüssel in dieselbe Liste (Liste `5`) eingefügt werden. Am besten ist es, wenn eine Hashfunktion alle Schlüssel möglichst gleichmäßig auf alle Listen der Hash-Tabelle verteilt.

Aufgabe: Geben Sie 14 Schlüssel (möglichst bekannte weibliche und männliche Vornamen) an, die von der Hashfunktion `hash03` möglichst gleichmäßig auf die 10 Listen der Hash-Tabelle verteilt werden (dabei heißt "möglichst gleichmäßig": pro Liste 1 oder 2 Schlüssel).

Die folgende Funktion `hash04` ist für die 14 Beispiel-Schlüssel besser als `hash03`:

```
14  int hash04(String s) {
15     // Der Index wird aus 3 Zeichen von s berechnet (dem ersten und dem
16     // letzten Zeichen und einem Zeichen aus der Mitte):
17     int vorn   = s.charAt(0)          % 3;
18     int mitte  = s.charAt(s.size()/2) % 5;
19     int hinten = s.charAt(s.size()-1) % 7;
20     return (vorn + mitte + hinten)    % LAENGE_DER_HASHTAB;
21  }
```

Hier die mit Hilfe der Hashfunktion `hash04` gefüllte Hash-Tab:

```
Index| Elemente in den Listen
  0  |
  1  |
  2  | Susi
  3  | Bine
  4  | Alex
  5  | Ali Babsy Alice Max
  6  | Benno Franz
  7  | Angy
  8  | Alfred Arno Kurt
  9  | Alf
Suchschritte insgesamt mit hash04: 24
```

Man sieht: hier sind nur noch *2* der 10 Listen *leer* und die längsten Listen enthalten *4* Schlüssel. Die folgende Hashfunktion ist für die 14 Beispiel-Schlüssel noch etwas *besser*:

```
22  int hash05(string s) {
23     // Der Ergebnis-Index wird im wesentlichen aus der Summe aller Zeichen
24     // des Schluessels s berechnet:
25     int i = 0;
26     for (int j=0; j<s.size(); j++) {
27        i += s[j] % 32 + j;
28     }
29     return i % LAENGE_DER_HASHTAB;
30  }
```

Der Ausdruck `s[j] % 32` bezeichnet die rechten 6 Bits des j-ten Zeichens von s. Bei den lateinischen Buchstaben (A-Z und a-z) spielen alle anderen Bits keine Rolle und sind gleich 0. Die Funktion `hash05` verteilt die 14 Beispiel-Schlüssel wie folgt auf die 10 Listen unserer Hash-Tabelle:

```
Index| Elemente in den Listen
  0  | Alice Benno
  1  | Alfred Max
  2  | Alf
  3  | Angy
  4  | Arno Susi
  5  | Ali Franz
  6  | Kurt Bine
  7  |
  8  | Alex
  9  | Babsy
Suchschritte insgesamt mit hashFunk05: 19
```

Das ist schon nahe am Optimum: Nur noch *eine* Liste ist leer geblieben und keine Liste enthält mehr als 2 Schlüssel.

Hash-Tabellen mit guten Hashfunktionen haben folgende angenehme Eigenschaft: Wenn man die Hash-Tabelle (d.h. die Reihung) vergrößert, werden die einzelnen Listen im allgemeinen kürzer und das Einfügen und Suchen wird schneller. Allerdings vergrößert man gleichzeitig die Wahrscheinlichkeit, dass einige oder viele der Listen leer bleiben und die entsprechende Null-Referenz nur Speicherplatz vergeudet. Manche Programme prüfen deshalb vor der Erzeugung einer Hash-Tabelle, wie viel freier Speicher vorhanden ist und berechnen daraus eine "gute Größe" für die Hash-Tabelle. Diese Tabelle ist umso schneller, je mehr freier Speicher vorhanden war. Bei wenig freiem Speicher funktioniert die Tabelle auch noch, ist aber entsprechend langsamer.

Eine weitere typische und angenehme Eigenschaft einer guten Hash-Tabelle: Solange die einzelnen Listen sehr kurz bleiben, ist die Zeit für den Zugriff auf ein Objekt (sowohl beim Einfügen als auch beim Suchen) praktisch unabhängig von der Größe der Tabelle (d.h. von der Länge der Reihung und von der Anzahl der eingefügten Objekte). Zum Vergleich: Beim *binären Suchen* in einer sortierten

Reihung wächst die Zeit für das Suchen mit dem Logarithmus der Länge der Reihung (damit wächst sie zwar langsam, aber sie wächst).

Hashfunktionen werden in der Praxis typischerweise von Spezialisten mit fundierten Kenntnissen in Statistik entwickelt. Diese Spezialisten versuchen, möglichst viel über die statistischen Eigenschaften der Schlüssel herauszufinden und die Hashfunktion auf diese Eigenschaften abzustimmen. Je mehr statistische Eigenschaften der Schlüssel einem bekannt sind, desto besser kann man die Hashfunktion darauf abstimmen. Wenn man z.B. weiss, dass das erste Zeichen eines Schlüssels immer ein Buchstabe ist, dann wird man wahrscheinlich von diesem ersten Zeichen nur die Bits verwenden, durch die sich Buchstaben voneinander unterscheiden. Und sollte man sogar wissen, dass als erstes Zeichen z.B. nur `'A'` und `'B'` in Frage kommen, dann nimmt man nur das Bit, durch das die beiden Zeichen sich unterscheiden etc. Bei sehr grossen Schlüsseln besteht die Kunst darin, "die wichtigen Bits" herauszufinden und alle anderen Bits unberücksichtigt zu lassen, damit die Hashfunktion schnell ist.

In Java besitzt jedes Objekt ob eine Methode namens `hashCode`, die einen `int`-Wert liefert. Mit Hilfe der Operation `ob.hashCode() % LAENGE_DER_HASHTABELLE` kann man diese "universelle" Hashfunktion an die Länge einer bestimmten Hash-Tabelle anpassen. Die Methode `hashCode()` wurde von Spezialisten entwickelt, berücksichtigt aber natürliche nicht die besonderen Eigenschaften einer bestimmten Population von Schlüsseln. Deshalb muss man auch in Java für bestimmte Anwendungen maßgeschneiderte Hashfunktionen entwickeln lassen. Hier die Ausgabe eines Java-Programms, in dem die Methode `hashCode` zum Einfügen in eine Hash-Tabelle der Länge 10 verwendet wurde:

```
Index|Elemente der Listen
  0  | Alfred Arno Susi
  1  |
  2  | Kurt Bine
  3  | Franz
  4  | Alex Max
  5  | Alf
  6  |
  7  | Babsy
  8  | Ali Alice Benno
  9  | Angy
Mit hashCode()-als-Hashfunktion Anzahl Suchschritte insgesamt: 22
```

Aufgabe: Schreiben Sie eine Klasse `HashTab<S, K>`, die die Schnittstelle `SammlungG<S, K>` implementiert und die Sammlung als *Hashtabelle* realisiert. Besonders interessant (und erstaunlicherweise auch einfach) wird Ihre Lösung, wenn Sie die Klasse `VerketteteListe<S, K>` aus einer früheren Aufgabe (siehe oben) verwenden und die Hashtabelle als Reihung von `VerketteteListe`-Objekten vereinbaren.

7. Komplexität eines Algorithmus

Ein *Algorithmus* ist eine Beschreibung die angibt, wie man ein bestimmtes (algorithmisches) Problem lösen kann. Hier ein paar berühmte Beispiele:

1. Der griechische Mathematiker *Euklid* (der im 3. Jahrhundert vor Christus lebte) hat in seinem Buch "Elemente der Mathematik" einen Algorithmus zur Berechnung des *größten gemeinsamen Teilers* (`ggT`) zweier natürlicher Zahlen beschrieben. Vermutlich war dieser Algorithmus auch vielen griechischen Handwerkern der damaligen Zeit (insbesondere den Fliesenlegern) bekannt.

Beispiele: der `ggT` von `6` und `10` ist gleich `2`, `ggT(15, 6)` ist gleich `3`, `ggT(12, 20)` ist gleich `4`, `ggT(7, 13)` ist gleich `1` etc.

2. Nach dem griechischen Mathematiker *Eratosthenes* (276 bis 197 vor Christus) ist ein Algorithmus zum Berechnen aller *Primzahlen* in einem bestimmten Zahlenbereich 1 bis n benannt ("Das Sieb des Eratosthenes").

3. Der Algorithmus *quicksort* zum Sortieren von Reihungen.

4. Der *Knuth-Morris-Pratt-Algorithmus* zum Suchen einer (meist kürzeren) Zeichenkette in einem (meist längeren) Text.

Algorithmen können in Form *konkreter Programme* in einer bestimmten Programmsprache vorliegen, werden aber häufig *abstrakter* dargestellt, z.B. in einer *algorithmischen Sprache*, die nicht unbedingt auf einem Rechner implementiert ist oder sogar in einer *natürlichen Sprache* (Griechisch, Türkisch, Deutsch, Englisch, ...). Jede Darstellung eines Algorithmus sollte aber so genau und klar sein, dass man sie ohne grosse Probleme in eine beliebige konkrete Programmiersprache übertragen kann.

Hier ein Beispiel für einen Algorithmus, dargestellt als Java-Unterprogramm:

```
1      static int summe(final int N) {
2         // N sollte groesser oder gleich 1 sein. Liefert die Summe aller
3         // Ganzzahlen von 1 bis N.
4
5         int erg = 0;
6         for (int zahl=1; zahl<=N; zahl++) {
7            erg = erg + zahl;
8         }
9         return erg;
10     } // summe
```

Wie schnell oder langsam ist dieser Algorithmus? Um diese Frage zu beantworten, könnte man das Unterprogramm `summe` in ein geeignetes Hauptprogramm einbauen, es mit verschiedenen Parametern aufrufen und *konkrete Zeitmessungen* vornehmen.

Die Ergebnisse solcher Zeitmessungen wären allerdings nicht nur von dem abstrakten *Algorithmus* abhängig, der von dem Unterprogramm `summe` realisiert wird, sondern auch von seiner konkreten *Implementierung*. Zur Implementierung gehört der verwendete *Rechner* (die Hardware), die *Programmiersprache* (im Beispiel: Java) und die *unterstützende Software* (Compiler, Interpreter, Betriebssystem etc.). Wenn wir denselben Algorithmus z.B. in *C* programmieren oder in ein paar Wochen auf einem neuen Rechner ausführen lassen, bekommen wir wahrscheinlich ganz andere Messergebnisse.

Problem: Können wir irgendetwas über die "Geschwindigkeit" eines abstrakten Algorithmus herausfinden, unabhängig von seinen konkreten Implementierungen in verschiedenen Sprachen, auf verschiedenen Rechnern etc. ?
Können wir z.B. etwas über den Algorithmus `summe` herausfinden, was auch in 10 oder 20 Jahren noch gilt und was selbst dann noch gültig bleibt, wenn wir den Algorithmus statt in Java in einer anderen Programmiersprache implementieren? Versuchen wir es mal:

Der Algorithmus `summe` schreibt dem Ausführer vor, zur Berechnung von `summe(N1)` im Wesentlichen `N1` viele *Schritte* durchzuführen. Bei heute üblichen Implementierungen besteht dabei *ein Schritt* aus zwei Additionen (`erg = erg + zahl;` und `zahl++`), einem Vergleich (`zahl > N`) und einem Sprungbefehl (zum Anfang der Schleife bzw. zum Beenden der Schleife). Vermutlich wird auch in Zukunft gelten:

Wenn wir 1.) `summe(N1)` und 2.) `summe(2*N1)` berechnen lassen, muss der Ausführer für die *zweite* Berechnung etwa *doppelt soviele Schritte* durchführen wie für die *erste* Berechnung.

Und wenn die Ausführung *eines Schritts* (unabhängig von der Größe der beteiligten Zahlen) immer *gleich viel Zeit* beansprucht, dann gilt:

Um nach dem obigen Algorithmus `summe(2*N1)` zu berechnen ist etwa *doppelt soviel Zeit* erforderlich wie zur Berechnung von `summe(N1)`.

Aufgabe: Betrachten Sie den Algorithmus *Bubblesort* (zum Sortieren einer Reihung von `int`-Werten), dargestellt durch das folgende Java-Unterprogramm:

```
1      static void bubbleSort(int[] ir) {
2         // Sortiert die Reihung ir aufsteigend.
3         int tmp;
4         for (int j=0; j<ir.length; j++) {
5            for (int i=0; i<ir.length-1; i++) {
6               if (ir[i] > ir[i+1]) {
7                  tmp     = ir[i];
8                  ir[i]   = ir[i+1];
9                  ir[i+1] = tmp;
10              }
11           } // for i
12        } // for j
13     } // bubbleSort
```

Wie oft muss die `if`-Anweisung (in Zeile 6 bis 10) ausgeführt werden, um eine Reihung der Länge `10` zu sortieren? Für eine Reihung der Länge `100`? Allgemein: Für eine Reihung der Länge `N`? Wie verändert sich die Anzahl der `if`-Ausführungen, wenn man die Länge der Reihung `ir` *ver-2-facht, ver-3-facht, ..., ver-n-facht*?

Hier eine Variante des Algorithmus `summe`, bei dem (praktisch beliebig grosse) Ganzzahlen des Typs `BigInteger` addiert werden:

```
1      static BigInteger summeBig(final BigInteger N) {
2         // N sollte groesser oder gleich 1 sein. Liefert die Summe aller
3         // Ganzzahlen von 1 bis N.
4
5         BigInteger erg  = BigInteger.ZERO;
6         for (
7            BigInteger zahl = BigInteger.ONE;
8            zahl.compareTo(N)<=0;
9            zahl = zahl.add(BigInteger.ONE)
10        )
11        {
12           erg = erg.add(zahl);
13        }
14        return erg;
15     } // summeBig
16
```

Bei diesem Algorithmus ist es *nicht plausibel* anzunehmen, dass jede Addition *ein* Schritt ist, dessen Ausführung immer *gleich lange* dauert.

Wenn Sie Ganzzahlen "mit Papier und Bleistift" addieren, wieviele Schritte brauchen Sie dann, um zwei Zahlen der Länge Z zu addieren? Mit der *Länge* einer Zahl ist hier die Anzahl ihrer *Dezimalziffern* gemeint.

Aufgabe: Wie kann man aus einer Ganzzahl *N* die Anzahl *Z* ihrer Ziffern im 10-er-System (Dezimalziffern) berechnen? Die Anzahl ihrer Ziffern im 2-er-System? Die Anzahl ihrer Ziffern im 17-er-System?

Mit einem *Algorithmus* kann man *Probleme verschiedener Größen* lösen. Die Größe der Probleme kann man in aller Regel durch *eine* natürliche Zahl *N* beschreiben.

Beispiele: Mit dem Algorithmus `bubbleSort` kann man kürzere und längere Reihungen sortieren. Als Maß für die Problemgröße nehmen wir die *Länge der* zu sortierenden *Reihung*.

Beispiel: Mit dem Algorithmus `summe` (bzw. `summeBig`) kann man die Summe aller Ganzzahlen von `1` bis (zu einer wählbaren Zahl) `N` berechnen. Als Maß für die Problemgröße nehmen wir die Zahl `N`.

Die (Zeit-) *Komplexität* eines Algorithmus drückt aus, wie die Anzahl der erforderlichen *Lösungsschritte* sich mit der *Problemgröße* `N` verändert.

Beispiel: Der Algorithmus `summe` hat eine Komplexität von `O(N)`. Das bedeutet: Wenn man die Problemgröße um den Faktor *2*, *3*, ..., *n* vergrößert, dann vergrößert sich auch die Anzahl der erforderlichen Lösungsschritte um den Faktor *2*, *3*, ..., *n*.

Beispiel: Der Algorithmus `bubbleSort` hat eine Komplexität von $O(N^2)$. Das bedeutet: Wenn man die Problemgröße um den Faktor *2*, *3*, ..., *n* vergrößert, dann vergrößert sich die Anzahl der erforderlichen Lösungsschritte um den Faktor 2^2, 3^2, ..., n^2.

Beispiel: Der Algorithmus *summeBig* hat eine Komplexität von `O(N * log(N))`. D.h. die Anzahl der erforderlichen Lösungsschritte wächst schneller als bei einem `O(N)`-Algorithmus, aber (deutlich) langsamer als bei einem $O(N^2)$-Algorithmus (weil die Funktion `F = log(N)` deutlich langsamer wächst als die Funktion `F = N`).

8. Aufgaben zur Zeitkomplexität

Nehmen Sie an, dass die Prozedur `blubs` einen *konstanten Zeitbedarf* hat (d.h. jeder Aufruf von `blubs` benötigt für seine Ausführung gleich viel Zeit, z.B. 37 Millisekunde). Geben Sie für jede der folgenden Prozeduren `p1`, `p2`, ..., `p9` an:

1. wie sich ihr Zeitbedarf verändert, wenn man den Parameter *N verdoppelt* und
2. welche *Zeitkomplexität* die Prozedur hat (z.B. `O(N)` oder $O(N^2)$ oder $O(N^3)$ oder `O(N* log N)` etc.).

Gehen Sie davon aus, dass die Prozeduren nur mit *positiven Parametern* aufgerufen werden (und Aufrufe wie z.B. `p1(-3)` oder `p5(-17)` etc. *nicht vorkommen*).

```
1   static void p1(final int N) {
2      for (int i=0; i<N; i++) {
3         blubs();
4      }
5   } // p1
6
7   static void p2(final int N) {
8      for (int i=0; i<N; i++) {
9         for (int j=0; j<N; j++) {
10           blubs();
11        }
12     }
13  } // p2
```

```
14
15  static void p3(final int N) {
16     for (int i=0; i<N; i++) {
17        blubs();
18     }
19     for (int i=0; i<N; i++) {
20        blubs();
21     }
22  } // p3
23
24  static void p4(final int N) {
25     for (int i=0; i<N; i++) {
26        for (int j=0; j<N; j++) {
27           for (int k=0; k<N; k++) {
28              blubs();
29           }
30        }
31     }
32  } // p4
33
34  static void p5(final int N) {
35     for (int i=0; i<N; i++) {
36        for (int j=0; j<i; j++) {
37           blubs();
38        }
39     }
40  } // p5
41
42  static void p6(final int N) {
43     for (int i=0; i<N/2; i++) {
44        for (int j=0; j<N/4; j++) {
45           for (int k=0; k<N/8; k++) {
46              blubs();
47           }
48        }
49     }
50  } // p6
51
52  static void p7(final int N) {
53     for (int i=0; i<N*N; i++) {
54        for (int j=0; j<N*N*N; j++) {
55           blubs();
56        }
57     }
58  } // p7
59
60  static void p8(final int N) {
61     blubs();
62     if (N>1) p8(N-1);
63  } // p8
64
65  static void p9(final int N) {
66     blubs();
67     if (N>1) {
68        p9(N-1);
69        p9(N-1);
70     }
71  } // p9
```