Algorithmen und Datenstrukturen
Übungsblatt 06

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungsblattbetreuer:	Nils Purschke
SoSe 2022	v1.0.2
Themen:	Hashtabellen
Relevante Foliensätze:	Foliensätze/Video/Zusammenfassung zu Hashtabellen
Abgabe der Hausübung:	01.07.2022 bis 23:50 Uhr

Hausübung 06 — Gesamt: 32 Punkte

Hashtabellen

Verbindliche Anforderungen für alle Hausübungen:

Das Dokumentieren und Kommentieren Ihres Quelltextes ist nicht verbindlich, wird jedoch zum besseren Verständnis Ihrer Lösung dringend empfohlen. Alle zur Bewertung dieser Hausübung relevanten Deklarationen von Klassen, Methoden (hierzu zählen auch Konstruktoren) und Attributen sind bereits in der Quelltext-Vorlage enthalten und dürfen nicht modifiziert oder entfernt werden. Ihnen steht aber frei, Hilfskonstrukte in Form von weiteren Klassen, Methoden und Attributen zu erstellen, sofern dies nicht explizit auf dem Übungsblatt verboten wurde und Ihre Hilfskonstrukte nicht gegen verbindliche Anforderungen verstoßen. Datenstrukturen und Hilfsmethoden aus der Java-Standardbibliothek sowie Arrays dürfen nicht verwendet werden, sofern dies nicht explizit auf dem Übungsblatt gefordert oder erlaubt wurde. Ihre Methoden müssen auch dann funktionieren, wenn Aufrufe von in der Vorlage deklarierten Methoden (auch von solchen, die von Ihnen implementiert werden) durch andere, korrekte Implementationen ersetzt werden.

Der Verstoß gegen verbindliche Anforderungen führt zu Punktabzügen und kann die korrekte Bewertung Ihrer Abgabe unter Umständen beeinflussen. Die Implementation einer in der Quelltext-Vorlage deklarierten Methode wird nur bewertet, wenn der mit `TODO` markierte Exception-Wurf entfernt wird.

Hinweise für alle Hausübungen:

Die zu verwendenen Zugriffsmodifizierer sind in der Vorlage bereits gegeben und werden auf dem Übungsblatt nicht immer angegeben. Beachten Sie weiter die Seiten *Informationen zum Importieren, Bearbeiten und Exportieren von Hausübungen* sowie *Informationen zu Plagiarismus* im Moodle-Kurs.

Bei Fragen stehen wir Ihnen vorzugsweise im Moodle-Kurs und in den Sprechstunden zur Verfügung.

Die für diese Hausübung in der Vorlage relevanten Verzeichnisse sind `src/main/h06` und `src/test/h06`.

Einleitung

In diesem Übungsblatt werden Sie Hashtabellen realisieren und die verschiedenen Möglichkeiten dazu bezüglich Effizienz miteinander vergleichen.

Klassen und Interfaces aus der Java-Standardbibliothek, die hier erwähnt werden, finden Sie in den Packages `java.lang`, `java.util` und `java.util.function`.

Anmerkung:

Per Konvention wird im Englischen die mit einem Schlüsselwert assoziierte Information mit dem Wort *value* bezeichnet. Im deutschsprachigen Fließtext wird im Folgenden das Wort *Information* verwendet, im Code und in Kommentaren zum Code verwenden wir und auch Sie stattdessen das Wort `value`.

Definition — Modulare Arithmetik Sie werden an einzelnen Stellen eine grundlegende Art von Arithmetik anwenden müssen: *modulare Arithmetik*. Damit lassen sich Fragen der Art beantworten: was wird die Uhrzeit (genauer: die momentane Stunde $\in \{0, \ldots, 23\}$) in genau 1000 Stunden sein bzw. was war die Uhrzeit vor genau 1000 Stunden.[a] Kennzeichnend ist: Wenn $h_1 - h_2$ durch 24 teilbar ist, gilt $h_1 \bmod 24 = h_2 \bmod 24$, also dieselbe Stunde des Tages (im Fall $h_1 \neq h_2$ natürlich verschiedene Tage).
In der ersten Version der 1000-Stunden-Frage oben sind beide Operanden, der Dividend $h + 1000$ und der Divisor 24, positiv. In diesem Fall bekommt man mit $(h+1000)\%24$ die korrekte Antwort, wobei $h \in \{0, \ldots, 23\}$ die Stunde ist, auf die sich die Frage bezieht, zum Beispiel $h = 19$: $1019\%24 = (1019 - 42 \cdot 24)\%24 = 11\%24 = 11$.
In der zweiten Version der 1000-Stunden-Frage oben ist der Dividend hingegen negativ: $h - 1000$. In diesem Fall liefert `'%'` ***nicht*** die korrekte Antwort. Den Grund finden Sie in Kapitel 01b, ab Folie 248 der FOP: [b] Division mit Rest rundet immer zur 0 hin. Bei negativem Dividenden heißt das also: Es wird ***auf*** gerundet. Wieder Beispiel $h = 19$: Die richtige Antwort lautet $(19 - 1000) \bmod 24 = (-981) \bmod 24 = (-981 + 41 \cdot 24) \bmod 24 = 3 \bmod 24 = 3$. Mit `'%'` kommt hingegen die falsche Antwort 21 heraus, weil zur nächsten Mitternacht ***auf***- und nicht zur letzten Mitternacht ***ab***gerundet wird.
Daher sollten Sie nicht `'/'` und `'%'` für modulare Arithmetik nutzen (oder, falls doch, sehr genau wissen, was Sie tun). Stattdessen bietet Klasse `Math` die beiden Methoden `floorDiv` und `floorMod` (die Namen kommen natürlich daher, dass immer abgerundet wird). Die Oracle-Dokumentation zu den beiden Methoden erläutert die Unterschiede zu `'/'` und `'%'` auch nochmals.

[a] Selbstverständlich gibt es auch gewichtigere Anwendungen der modularen Arithmetik, zum Beispiel in der Kryptographie.
[b] ***Achtung***: In früheren Versionen des Foliensatzes FOP 01b fehlte dieser Abschnitt noch!

Eine Kurzzusammenfassung des obigen Absatzes über modulare Arithmetik: Anstelle von `'%'` verwenden Sie bitte `floorMod`. Details dazu oben.

H1: Basisdatenstrukturen 5 Punkte

Bereits in der Vorlage vorhanden sind zwei generische Interfaces, `Fct2Int` und `BinaryFct2Int`, beide mit generischem Typparameter `T`. Analog zu den diversen `Function`-Interfaces in der Java-Standardbibliothek haben beide Interfaces eine Objektmethode `apply`. In beiden Interfaces hat diese Methode `apply` einen Parameter vom formalen Typ `T` und liefert `int` zurück. Bei `BinaryFct2Int` hat sie noch einen zweiten Parameter vom formalen Typ `int`.

Beide Interfaces, `Fct2Int` und `BinaryFct2Int`, haben jeweils noch das übliche `get/set`-Methodenpaar mit Namen `getTableSize` und `setTableSize` für ein Attribut `tableSize` vom formalen Typ `int`, das in den implementierenden Klassen vorhanden ist.

Vervollständigen Sie die generische `public`-Klasse `Hash2IndexFct` mit generischem Typparameter `T`, die `Fct2Int<T>` implementiert. Der `public`-Konstruktor hat als ersten Parameter `initialTableSize` und als zweiten Parameter `offset`, beide vom formalen Typ `int`. Der aktuale Wert von `offset` wird vom Konstruktor in einer geeigneten `private`-Objektkonstanten gespeichert. Die `private`-Objektvariable `tableSize` wird im Konstruktor mit `initialTableSize` initialisiert. Auf diese Objektvariable greifen `get-/setTableSize` zu. Sie dürfen davon ausgehen, dass `initialTableSize` größer gleich 2 ist.

Die Methode `apply` von `Hash2IndexFct` soll den Rest einer ganzzahligen Division mittels `Math.floorMod` zurückliefern. Dividend dieser ganzzahligen Division ist die Summe aus zwei Werten: der Betrag (mathematische Betrag) des Hashcodes des aktualen Parameterwertes [1] und der Wert der Objektkonstante, die durch den Parameter `offset` gesetzt wurde. Divisor ist der momentane Wert der `tableSize`. Das Ergebnis der Restwertbildung ist also im Indexbereich eines Arrays der Größe `tableSize`.

Vervollständigen Sie die Methode `apply` in Klasse `LinearProbing` anhand folgender Beschreibung: Seien x und i die aktualen Parameterwerte beim Aufruf der Methode `apply` von `LinearProbing` und sei a die Rückgabe der Methode `apply` des Attributs bei Aufruf mit x; dann ist $(a + i)$mod `tableSize` die Rückgabe der Methode `apply` von `LinearProbing`. Sie können davon ausgehen, dass $a >= 0$ und $i >= 0$ gilt.

Die Rückgabe der Methode `apply` von `DoubleHashing` ist mathematisch folgendermaßen definiert: Seien x und i die aktualen Parameterwerte der Methode `apply` von `DoubleHashing` und seien a bzw. b die Rückgaben der Methode `apply` von `fct1` bzw. `fct2` bei Aufruf mit x; dann ist $(a + i \cdot b)$mod `tableSize` die Rückgabe der Methode `apply` von `DoubleHashing`. Sie können davon ausgehen, dass $a >= 0$, $b >= 0$ und $i >= 0$ gilt. Sollte b eine gerade Zahl sein, so erhöhen Sie b **vor** der Berechnung um eins, sodass b ungerade wird.

Dies hat folgenden Hintergrund: Für eine Hashfunktion $h(k, i) = h0(k) + i \cdot h1(k)$ mod m, müssen, falls alle Indizes durchlaufen werden sollen, $h1(k)$ und m für alle k teilerfremd sein. Also für alle k gilt: $ggT(h1(k), m) = 1$. Beispiele:

- m = Zweierpotenz und $h1(k)$ immer ungerade.
- m = Primzahl und $0 < h1(k) < m$.

Auch muss für alle k gelten, dass $h1(k) \neq 0$.

Verbindliche Anforderung:
Stellen Sie sicher, dass es bei den Berechnungen zu keinen arithmetischen Überläufen kommt. Fallen Ihnen geeignete mathematische Umformung (auch hier kommt natürlich wieder modulare Arithmetik ins Spiel) und oder Typkonvertierung ein? Denken Sie auch an Extremfälle wie `tableSize = Integer.MAX_VALUE` und `a = b = i = Integer.MAX_VALUE - 1`.

[1] Den Hashcode eines Objekts erhalten Sie durch Aufruf der Methode `hashCode()`.

H2: Hashtabellen

Bereits in der Vorlage enthalten ist ein generisches Interface `MyMap` mit generischen Typparametern `K` und `V` (`K` = *key type* = Schlüsselwerttyp und `V` = *value type* = Informationstyp). Die Darstellungsinvariante von `MyMap` ist:

1. Ein `MyMap`-Objekt repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine endliche Menge von Paaren (`key`,`value`), die auch leer sein kann. Diese Menge kann sich über die Zeit hinweg beliebig häufig, aber ausschließlich durch Aufruf der Methoden von `MyMap` ändern.
2. Jeder *Schlüsselwert* `key` ist vom formalen Typ `K` und ungleich `null`.
3. Kein Schlüsselwert kommt in einem `MyMap`-Objekt zu irgendeinem Zeitpunkt zweimal vor, das heißt, für keine zwei im selben Moment im selben `MyMap`-Objekt repräsentierten Schlüsselwerte `key1` und `key2` gilt `key1.equals(key2)`.
4. Der mit `key` assoziierte `value` ist vom formalen Typ `V` und ungleich `null`.

Interface `MyMap` hat folgende Methoden:

- `containsKey`: hat einen Parameter `key` vom formalen Typ `K` und Rückgabetyp `boolean`; liefert genau dann `true` zurück, wenn es ein Paar (`key`,`value`) in dem `MyMap`-Objekt gibt, mit dem `containsKey` aufgerufen wird.
- `getValue`: hat einen Parameter `key` vom formalen Typ `K` und Rückgabetyp `V`; falls `key` momentan im `MyMap`-Objekt enthalten, also irgendein Wert `value` vom formalen Typ `V` momentan mit `key` assoziiert ist, wird `value` zurückgeliefert; ansonsten wird `null` zurückgeliefert.
- `put`: der erste Parameter, `key`, ist vom formalen Typ `K`, der zweite Parameter ist vom formalen Typ `V`; Rückgabetyp ist `V`; falls `key` unmittelbar vor Aufruf von `put` im `MyMap`-Objekt enthalten war, wird die bisher mit `key` assoziierte Information zurückgeliefert, und der zweite aktuale Parameter von `put` wird die neue mit `key` assoziierte Information; andernfalls wird ein neues (`K`, `V`)-Paar bestehend aus den beiden aktualen Parametern von `put` in das `MyMap`-Objekt eingefügt und `null` zurückgeliefert. Intuitive Formulierung: Neues (Schlüssel, Wert) Paar einfügen. Gegebenenfalls bestehendes Wert mit gleichem Schlüssel ersetzen, dabei alten Wert zurückgeben.
- `remove`: hat einen Parameter `key` vom formalen Typ `K` und Rückgabetyp `V`; falls `key` in dem `MyMap`-Objekt, mit dem `remove` aufgerufen wird, enthalten ist, werden `key` und die damit assoziierte Information aus dem `MyMap`-Objekt entfernt und letztere zurückgeliefert; andernfalls wird der Inhalt des `MyMap`-Objektes nicht verändert und `null` zurückgeliefert.

H3: Mehrfachsondierung 10 Punkte

Vervollständigen Sie die Klasse `MyIndexHoppingHashMap`. Diese hat drei Objektvariablen: `theKeys` vom Typ „Array von `K`“, `theValues` vom Typ „Array von `V`“ und `occupiedSinceLastRehash` vom Typ „Array von `boolean`“. Wesentliche Implementationsinvariante ist, dass ein Objekt von Klasse `MyIndexHoppingHashMap` zu jedem Zeitpunkt nach Beendigung des Konstruktors die folgenden Bedingungen erfüllt:

1. Die drei Arrays `theKeys`, `theValues` und `occupiedSinceLastRehash` sind nicht `null` und haben dieselbe Länge.

2. Die vom `MyIndexHoppingHashMap`-Objekt repräsentierten Schlüsselwerte sind genau die in `theKeys[`i`]` gespeicherten Werte, also die $i \in \{0,\ldots,$`theKeys.length`$-1\}$ und `theKeys[`i`] != null`. Die zu einem Schlüsselwert `theKeys[`i`]` gespeichete Information ist `theValues[`i`]`.

3. Für $i \in \{0,\ldots,$`theKeys.length`$-1\}$ gilt: falls `theValues[`i`] != null`, dann auch `theKeys[`i`] != null`. Mit anderen Worten: Es gibt keine Information in `theValues`, die nicht mit einem Schlüsselwert in `theKeys` assoziiert ist.

4. Für $i \in \{0,\ldots,$`theKeys.length`$-1\}$ ist der Wert in `occupiedSinceLastRehash[`i`]` genau dann `true`, wenn momentan `theKeys[`i`] != null` *ist* oder[2] `theKeys[`i`] != null` zu mindestens einem Zeitpunkt seit dem letzten Aufruf von `rehash` *war* bzw. – falls es noch keinen Aufruf von `rehash` gab – nach Beendigung des Konstruktors (Details von `rehash` weiter unten).

`MyIndexHoppingHashMap` protokolliert in der Objektvariable `occupiedCount` mit, wie viele Komponenten von `occupiedSinceLastRehash` momentan `true` sind. Der Konstruktor der Klasse `MyIndexHoppingHashMap` enthält vier Parameter:

1. Die initiale Länge der drei Arrays als `int` > 1, der Konstruktor richtet also alle drei Arrays mit dieser Größe ein.

2. Der Faktor als `double`-Zahl > 1, um den die Länge der drei Arrays bei jedem Aufruf von `rehash` (auf `int` abgerundet) wächst. Als Objektkonstante mit passendem Namen zu speichern.

3. Der Schwellwert für den Füll- und Fragmentierungsgrad als eine `double`-Zahl im offenen Intervall $(0\ldots1)$ als Objektkonstante mit passendem Namen zu speichern.

4. Das in `hashFunction` übergebene Objekt wird verwendet, um in den Methoden von `MyMap` die einzelnen Indizes zu berechnen, an denen gemäß Video/Folien zu Hashtabellen nachgeschaut wird, um den ersten aktualen Parameter bzw. eine unbesetzte Arraykomponente zu finden. Um welche Implementation es sich konkret handelt, darf für die Korrektheit Ihrer Lösung keinen Unterschied machen.

Die Idee hinter `occupiedSinceLastRehash` ist, dass auch ehemals belegte Plätze bei der Suche nicht zum Abbruch führen. Angenommen ein Element X wird gelöscht. Jetzt kann es beispielsweise sein, dass ein anderes Element Y an dieser Stelle (an der Stelle von Element X) vorher nicht eingefügt werden konnte, weil der Platz zu diesem Zeitpunkt noch belegt war. Nun ist der Platz zwar frei, da das Element X gelöscht wurde, vorher aber nicht. Damit bei der Suche klar ist, welche Plätze vormals belegt waren, gibt es das Array occupiedSinceLastRehash, in dem jenes festgehalten wird. Die Suche kann somit, sobald ein freier Platz gefunden wurde, der auch vormals nicht belegt war, terminieren, da an dieser Stelle sonst das Element eingefügt worden wäre. Es befindet sich entsprechend nicht in der Hashtabelle.

Die Methode `rehash` (Details unten) wird in `put` vor dem Einfügen des neuen `(key, value)`-Paares aufgerufen, falls durch das Einfügen des neuen `(key,value)`-Paares der Anteil derjenigen Indizes $i \in \{0,\ldots,$`theKeys.length`$-1\}$, für die `occupiedSinceLastRehash[`i`] == true` gilt, diesen Schwellwert (3. Parameter des Konstruktors) überschreiten würde. Oder anders formuliert, falls die Anzahl der Einträge im Array `occupiedSinceLastRehash` mit dem Wert `true`, NACH dem Einfügen größer als der Schwellwert $*$ `tableSize`, bzw. effektiv auch Schwellwert $*$

[2] Dieses „oder“ ist natürlich zu verstehen als ein inklusiv-oder.

`occupiedSinceLastRehash.length` WÄRE, so wird VOR dem Einfügen `rehash` aufgerufen ($*$ = MULTIPLIKATION).

Die Methode `apply` von `hashFunction` ist die Hashfunktion, mit der die Positionen der einzufügenden/zu löschenden/zu findenden Elemente bestimmt wird. Sie wird auf den Schlüsselwert vom Typ `K` angewendet (erster Parameter). Der zweite Parameter der Methode `apply` dient der Kollisionsvermeidung. Er wird immer dann erhöht, falls an der aktuellen Position eine Kollision beim Einfügen vorliegt oder das Element beim Löschen / Suchen an der aktuellen Position nicht gefunden wurde. Vor Durchführung jeder Operation (Suchen, Löschen, etc.) wird er darüber hinaus wieder auf 0 zurückgesetzt. Ist eine Arraykomponente i momentan unbesetzt, aber `occupiedSinceLastRehash[`i`]` ist `true`, dann suchen alle vier Methoden (`containsKey`, `getValue`, etc.) weiter, bis entweder der erste aktuale Parameter oder eine unbesetzte Arraykomponente j mit `occupiedSinceLastRehash[`j`] == false` erreicht ist. Falls Parameter `key` speziell bei Methode `put` nicht auf diesem Weg in `theKeys` gefunden wird, soll das Paar(`key`,`value`) im ersten gefundene leeren Index i abgespeichert werden, egal ob `occupiedSinceLastRehash[`i`] == true` oder `== false` ist. [3]

Die parameterlose Objektmethode `rehash` kreiert drei neue Arrays `theKeys`, `theValues` und `occupiedSinceLastRehash`, und zwar um so viel größer als die momentanen Arrays, wie die entsprechende Objektkonstante gemäß zweitem Parameter des Konstruktors besagt. Anders ausgedrückt entspricht die neue Größe, der aktuellen Größe, multipliziert mit `resizeFactor` (Ergebnis wird abgerundet). Alle Werte ungleich `null` in `this.theKeys` und `this.theValues` werden von vorne bis hinten (`for (int i = 0; i < n; i++)`) durch `rehash` mittels Zuweisungsoperator „=“ in die beiden neuen Arrays `theKeys` bzw. `theValues` kopiert, aber nicht einfach nur an denselben Indizes, an denen sie an den alten Arrays stehen. Stattdessen ersetzen Sie die alten Arrays durch die neuen (müssen aber natürlich kurzzeitig Verweise auf die alten speichern) und rufen `put` mit allen (`key`,`value`)-Paaren auf, die in den alten Arrays momentan gespeichert sind. Die Werte im neuen Array `occupiedSinceLastRehash` setzen Sie so, dass die obige Implementationsinvariante unmittelbar nach Beendigung des Aufrufs von `rehash` erfüllt ist. Die Größe der Hash-Tabelle in `BinaryFct2Int<K>` muss dann natürlich ebenfalls aktualisiert werden.

Ihre Aufgabe besteht nun darin die Methoden in Klasse `MyIndexHoppingHashMap` anhand des hier im Text beschriebenen Verhaltens umzusetzen.

> **Hinweis:**
>
> Eigentlich kann man ja in Java kein Array eines generischen Typs erzeugen (siehe Kapitel 06, ab Folie 138 der FOP). In Übungsblatt 02 haben wir aber einen Trick gesehen, mit dem das doch geht.

H4: Hashtabelle von Listen — 8 Punkte

Klasse `MyListsHashMap` hat eine Objektkonstante `table` vom Typ `LinkedList<KeyValuePair<K, V>>[]`. Die Klasse `KeyValuePair` ist bereits in der Vorlage komplett implementiert. Es kann allerdings nicht schaden sich die JavaDoc-Kommentare durchzulesen, um zu verstehen welche Funktion diese Klasse erfüllt.

Die wesentliche Implementationsinvariante von `MyListsHashMap` ist, dass zu jedem Zeitpunkt die Menge der in einem Objekt von `MyListsHashMap` gespeicherten (`key`,`value`)-Paare genau die Vereinigung der Mengen von Paaren in den einzelnen Komponenten des Arrays ist und dass kein (`key`,`value`)-Paar zweimal in dieser Vereinigung enthalten ist.

Methode `apply` von Attribut `hashFunction` liefert für einen Schlüsselwert den Index im Array zurück, an dem dieser Schlüsselwert in der dortigen Liste zu speichern ist bzw. nach der Speicherung wiedergefunden werden kann. Jedes Paar (`key`,`value`) wird beim Einfügen an Position 0 der Liste gespeichert. Dabei wird, insofern vorhanden, das bestehende Paar an Position 0 natürlich nicht überschrieben, dieses wandert an Position 1 der Liste. Das Paar, das vorher

[3] Dieser Fall kann natürlich nur dann auftreten, wenn der resizeFactor >= 1 ist, da ansonsten vorher ein rehash, einschließlich Vergrößerung der Hahstabelle, stattgefunden hätte. Er wird hier trotz der Einschränkung resizeFactor∈(0, 1) abgefangen.

an Position 1 war, insofern vorhanden, an Position 2 und so weiter. Sollte sich der Schlüssel (`key`) bereits in einem Paar (`key`, `value`) in der Liste befinden, so wird der assoziiert Wert (`value`) mittels `setValue` überschrieben, nicht aber die Position des Paars vom Typ `KeyValuePair` verändert. Sei `newValue` der einzufügende Value, dann hat das Paar nach dem Einfügen die gleiche Position innerhalb der Liste und folgende Form: (`key`, `newValue`).

Vervollständigen Sie in `MyListsHashMap` alle weiteren Methoden des Interfaces `MyMap` so, dass die Implementationsinvariante erfüllt bleibt.

H5: Eigene hashCode-Implementation — 3 Punkte

Vervollständigen Sie die `public`-Klasse `MyDate` mit fünf `private`-Objektkonstanten vom Typ `int`: Jahr 1970 . . . 2022, Monat, Tag im Monat, Stunde (24-Stunden-Format) und Minute. Der `public`-Konstruktor hat einen Parameter vom formalen Typ `java.util.Calendar` und initialisiert diese fünf Attribute durch die entsprechenden Werte im aktualen Parameter. Für jedes der fünf Attribute existiert eine `get`-Methode: `getYear`, `getMonth`, `getDay`, `getHour` und `getMinute`. Sie dürfen die fünf Objektkonstanten sowie Attribute intern auch als Array realisieren, wenn Sie das bevorzugen.

Ihre Klasse `MyDate` hat eine boolesche Objektkonstante sowie neben den oben genannten noch sechs weitere `private`-Objektkonstanten, alle sechs vom Typ `long`. Das boolesche Attribut wird durch einen zweiten, und zwar booleschen Parameter des Konstruktors initialisiert. Der Wert dieses Parameters kann über die `get`-Methode `getBool` abgefragt werden. Die sechs `long`-Attribute werden im Folgenden *Koeffizienten* genannt: fünf Koeffizienten für die fünf aus dem `Calendar`-Objekt gezogenen `int`-Attribute und ein Koeffizient für die Summe dieser fünf `int`-Attribute. Diese sechs `long`-Attribute werden ebenfalls im Konstruktor gesetzt, aber nicht auf Basis von Parametern des Konstruktors; Genaueres in der verbindlichen Anforderung unten.

Überschreiben Sie die in Klasse `Object` definierte Methode `hashCode` für `MyDate` so, dass die Rückgabe von `hashCode` im Prinzip wie folgt ist: Falls das boolesche Attribut `true` ist, soll jedes aus dem `Calendar`-Objekt gezogene Attribut mit seinem zugehörigen Koeffizienten multipliziert werden, und diese fünf Produkte werden zusammenaddiert. Ist hingegen das boolesche Attribut `false`, so werden die aus dem `Calendar`-Objekt gezogenen Attribute aufaddiert und die Summe mit dem sechsten Koeffizienten multipliziert. Da Methode `hashCode` Rückgabetyp `int` hat, müssen Sie das Ergebnis in jedem der beiden Fälle noch durch Modulo-Bildung mit dem größten durch `int` darstellbaren Wert (siehe Folien 16 ff. in Kapitel 11 der FOP) ermitteln. Überschreiben Sie des Weiteren die `equals` Methode so, dass Objekte mit gleichem `hashCode` Wert als gleich gelten. Dabei ist Voraussetzung, dass das andere Objekt auch vom Typ `MyDate` und nicht `null` ist. Ansonsten liefert `equals` den Wert `false` zurück.

Verbindliche Anforderung:
In Methode `hashCode` von `MyDate` sind alle arithmetischen Operationen im Datentyp `long` auszuführen. Keine dieser arithmetischen Operationen darf einen arithmetischen Überlauf erzeugen. Der Koeffizient für die Minuten lautet: 99998, der Koeffizient für die Stunden lautet: 1234, der Koeffizient für die Tage lautet: 4, der Koeffizient für die Monate lautet: 73232, der Koeffizient für die Jahre lautet: 4563766470487201 und der Koeffizient für die Summe lautet: 98924.

Unbewertete Verständnisfrage:
Die Konversion von `long` auf `int` könnte man auch mit dem Typecast Operator „`(int)`“ oder mit Methode `intValue` von `java.lang.Long` erreichen, aber dann müsste auf das Vorzeichen des Ergebnisses geachtet werden. Warum?

H6: Tests 6 Punkte

In dieser Aufgabe schreiben Sie einen Test, der die Laufzeit Ihrer Implementierung in bestimmten Szenarien ermitteln soll. Vervollständigen Sie dazu die `public`-Klasse `RuntimeTest`.

Die Klasse enthält eine `public static` Methode `generateTestdata` ohne Parameter und mit formalem Rückgabetyp `MyDate[][]`: Einzelne Testdaten werden generiert, indem Sie zufällige `long`-Werte aus einem `long`-Stream (siehe bspw. `Random` in der Java-Standardbibliothek) lesen und mit `Calendar.setTimeInMillis` aus jedem positiven `long`-Wert ein `Calendar`-Objekt erstellen, wobei dass das Jahr der damit erstellten `Calendar`-Objekte nicht größer als 2022 und kleiner als 1970 (Das Datum kann in Millisekunden ab 1970 angegeben werden) sein soll. Ihre Testsuite soll dabei insgesamt einen Satz von 1.000 Testdaten, jeweils bestehend aus einem einzelnen `Calendar`-Objekt, erstellen. WICHTIG: Generieren Sie nicht stumpf Zufallszahlen und verwerfen Sie alle Zufallszahlen < 0 oder Calendar-Objekte mit Jahr > 2022! Verwenden Sie stattdessen alle Zufallszahlen, indem Sie diese bspw. durch Modulo-Bildung auf ein entsprechendes Intervall begrenzen!

Aus jedem dieser `Calendar`-Objekte erstellen Sie dann zwei `MyDate`-Objekte, je eines mit Wert `true` bzw. `false`. Die Testdaten, die Sie mit `true` generiert haben, schreiben Sie in die erste Komponente des zweidimensionalen Arrays, welches die Methode zurückliefert. Die Testdaten, die Sie mit `false` generiert haben, schreiben Sie entsprechend in die zweite Komponente. Ihr Testdatensatz sollte somit aus einem zweidimensionalen Array bestehen, wobei das Haupt-Array die Länge 2 hat und die beiden untergeordneten Arrays die Länge der Testdaten (1.000) haben. Dies ist nun ihr Testdatensatz, auf dem Sie abschließend die Tests durchführen.

Um den Testsatz, der neben den Testdaten noch die entsprechende Hashtabelle enthält abzuspeichern, enthält die Vorlage die `public` Klasse `TestSet` mit formalem Typparameter `W`. `TestSet` enthält einen Konstruktor mit zwei Parametern: `hashTable` vom formalen Typ `MyMap<W,W>` (die für den Test zu nutzende Hashtabelle) und `testData` vom formalen Typ `W[]` (der Satz der Testdaten die für den Test genutzt werden). Zwei passende `private` Objektkonstanten speichern die Werte der Parameter. Über die get-Methoden `getHashTable` bzw. `getTestData` kann ihr Wert abgefragt werden. Set-Methoden existieren nicht.

Die Methode `public static createTestSet` mit Rückgabetyp `TestSet<MyDate>` erstellt den Testsatz: Die Parameter i, j, k, $\ell \in \{1, 2\}$ (in dieser Reihenfolge und vom formalen Typ `int`), sowie natürlich der Testdatensatz vom formalen Typ `MyDate[][]` als fünften Parameter nehmen dabei folgenden Einfluss:

- $i = 1$ heißt, diejenigen `MyDate`-Objekte bilden die Testmenge, bei deren Konstruktion der zweite aktuale Parameter `true` war; bei $i = 2$ entsprechend `false`.
- $j = 1$ heißt, dass als Hashtabelle die Klasse `MyIndexHoppingHashMap` benutzt wird, initialisiert mit Resize-Faktor(Parameter 2 des Konstruktors) von 2 und Resize-Schwellwert(Parameter 3 des Konstruktors) von 0,75 ; $j = 2$ heißt entsprechend, dass die Klasse `MyListsHashMap` benutzt wird. Die Hashtabelle akzeptiert in beiden Fällen nur Key und Value Werte vom formalen Typ `MyDate`, das heißt die Typparamter `K` und `V` sind beide `MyDate`.
- Für die dazugehörige passende Hashfunktion gilt: Bei MyIndexHoppingHashMap ($j = 1$) heißt $k = 1$, dass LinearProbing mit interner Hashfunktion `Hash2IndexFct` mit Offset 0 verwendet wird; $k = 2$ heißt, dass `DoubleHashing` mit interner Hashfunktion 1 `Hash2IndexFct` mit Offset 0 und interner Hashfunktion 2 `Hash2IndexFct` mit Offset 42 verwendet wird. Für MyListsHashMap ($j = 2$) ist k irrelevant und es wird die Hashfunktion `Hash2IndexFct` mit Offset 0 verwendet.
- $\ell = 1$ bzw. $\ell = 2$ heißt, die Hashtabelle ist **initial** sehr großzügig bzw. sehr klein im Verhältnis zur Größe der Testmenge, konkret: bei $j = 1$: die Anzahl der Komponenten der drei internen Arrays bei $\ell = 1$ ist 2^{12}, bei $\ell = 2$ ist die Anzahl 2^6; Für $j = 2$ gilt, dass die Anzahl der Komponenten in dem einen internen Array bei $\ell = 1$ dreimal so groß wie die Testmenge, bei $\ell = 2$ umfasst sie ein Zehntel der Größe der Testmenge.

Nach dem Setup, basierend auf den Parametern, ist sowohl eine voll initialisierte Hashtabelle, als auch der Testdatensatz im Objekt vom Typ `TestSet<MyDate>` vorhanden. Hierauf wird nun der eigentliche Test durchgeführt. Dieser Test wird über die `public static` Methode `test` (**Kein JUnit-Test**) mit Rückgabetyp `void` und Parameter testSet vom

formalen Typ `TestSet<MyDate>` durchgeführt. Er umfasst folgendes, wobei die Ergebnisse der einzelnen Operationen keine Relevanz haben und auch nicht ausgewertet oder gespeichert werden sollen:

1. Fügen sie die ersten 750 Elemente aus der Testmenge in die Hashtabelle ein (Key = Value = Element).
2. Überprüfen Sie für **alle** Elemente aus der Testmenge, ob diese in der Hashtabelle vorkommen.
3. Versuchen Sie für **alle** Elemente aus der Testmenge, den in der Hashtabelle gespeicherten Wert zu ermitteln.
4. Versuchen Sie für **alle** Elemente der Testmenge, diese aus der Hashtabelle zu löschen.

Die Ermittlung der Laufzeit der Tests und damit der Methode erfolgt extern und ist hier nicht weiter relevant. Die Geschwindigkeit ist aber kein Bewertungskriterium, Sie brauchen also keine besonderen Geschwindigkeitsoptimierungen an Ihrem Code durchzuführen.