Die Set-Implementationen im Detail

Variante 1: ArrayList

Die einfachste Lösung ist, wenn der ADT Set als interne Datenstruktur eine `ArrayList` verwendet. Beim Einfügen muss immer getestet werden, ob der Wert bereits in der `ArrayList` vorkommt. Dazu muss unter Umständen jedes Element der `ArrayList` untersucht werden.

Bewertung

Vorteile:

- Einfache Implementation
- Speicherbedarf relativ gering (etwa proportional zur Anzahl der Werte)

Nachteile:

- Suche nach einem Element bei großen Mengen aufwendig (proportional zur Anzahl der Werte)

Variante 2: Bitvektor

Man kann eine Menge von Integer-Zahlen auch in eine einzelne Integer-Zahl verpackt darstellen, indem man die entsprechenden Bits in ihrer Binärdarstellung setzt. Man spricht dann von einem **Bitvektor**.

Beispiel: Die Menge `{0,3,4}` soll gespeichert werden. Dazu setzt man die Bits an den Stellen 0,3 und 4 auf 1 und alle anderen auf `0` und bestimmt anschließend die Dezimaldarstellung der Binärzahl. Das Resultat ist die Zahl 25:

31	30	…	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	…	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	= 25

Da eine Integer-Zahl in Java 32 Bit lang ist, kann man mit einer Integer-Zahl also für die Zahlen von 0 bis 31 darstellen, ob sie in einer Menge enthalten sind oder nicht.

Ist eine Zahl in der Menge enthalten?

Um herauszufinden, ob eine Zahl enthalten ist, kann man mit dem binären UND-Operator prüfen, ob das entsprechende Bit gesetzt ist:

```
int vier_gesetzt = 25 & 16;  // Ergebnis: 16
```

```
int eins_gesetzt = 25 & 2;   // Ergebnis: 0
```

	31	30	...	8	7	6	5	4	3	2	1	0
25 =	0	0	...	0	0	0	0	1	1	0	0	1
16 =	0	0	...	0	0	0	0	1	0	0	0	0
2 =	0	0	...	0	0	0	0	0	0	0	1	0
25 & 16 =	0	0	...	0	0	0	0	1	0	0	0	0
25 & 2 =	0	0	...	0	0	0	0	0	0	0	0	0

Der binäre UND-Operator verknüpft zwei `int`-Zahlen (nach Umwandlung in die Binärdarstellung) und setzt im Ergebnis ein Bit auf 1, wenn die entsprechenden Bits in beiden Operanden auf 1 gesetzt sind.

Eine Zahl ist in der Menge enthalten, wenn bei der UND-Verknüpfung ein Wert herauskommt, der nicht 0 ist.

Zahlen hinzufügen

Um einen Wert zu einem Bitvektor hinzufügen, verwendet man die binäre ODER-Verknüpfung.

Beispiel: Zur Menge `{0,3,4}` soll die 6 hinzugefügt werden. Man setzt also den Bitvektor auf

```
bitvektor = bitvektor | 64; // neuer Wert 89
```

	31	30	...	8	7	6	5	4	3	2	1	0
25 =	0	0	...	0	0	0	0	1	1	0	0	1
64 =	0	0	...	0	0	1	0	0	0	0	0	0
25 | 64 =	0	0	...	0	0	1	0	1	1	0	0	1

Der binäre ODER-Operator verknüpft zwei `int`-Zahlen und setzt im Ergebnis ein Bit auf 1, wenn mindestens eines der entsprechenden Bits in den beiden Operanden auf 1 gesetzt ist.

Beim Setzen eines Bits muss nicht geprüft werden, ob es bereits gesetzt ist – der ODER-Operator verändert in diesem Fall den Bitvektor nicht.

Bitmasken

Zum lesen und setzen von Elementen benötigt man die entsprechenden Zahlen, letztlich in

Binärdarstellung. Diese Zahlen, die man zum Auslesen oder Setzen von Bits verwendet werden als `Bitmaske` bezeichnet.

Eine Maske, bei der als niederwertigstes Bit nur das n-te Bit von rechts gesetzt ist, entspricht der Zahl 2^n. Schnell und effizient erhält man diese Zahl, indem man die Zahl 1 um n Stellen nach links „verschiebt“.

In Java wird eine solche Bitverschiebung ("bit shift") nach links mit dem Operator « erreicht:

```
int y = 1 << 7;  // y ist  1*27 = 128
```

Man kann auch andere Zahlen als die 1 verschieben:

```
int x = 13 << 3; // x ist 13*23 = 104
```

	31	30	…	8	7	6	5	4	3	2	1	0
13 =	0	0	…	0	0	0	0	0	1	1	0	1
13 << 3 =	0	0	…	0	0	1	1	0	1	0	0	0
1 =	0	0	…	0	0	0	0	0	0	0	0	1
1 << 7 =	0	0	…	0	1	0	0	0	0	0	0	0

Zahlbereichserweiterung

Mit einer einzelnen 32-Bit-Zahl kann man Mengen der Zahlen 0 bis 31 abbilden. Will man größere Werte erlauben, benötigt man eine ArrayList von Integer-Zahlen.

Die k-te Zahl der ArrayList repräsentiert damit die Elemente mit den Werten 32·k bis 32·k+31.

	Index 0	Index 1	Index 2	…
daten	0100…1101101	1110…0110000	0001…0001101	…
	Bits 0 bis 31	Bits 32 bis 63	Bits 64 bis 95	…

Das Element n wird also durch das Bit n%32 in der Zahl `daten[n/32]` repräsentiert.

Beim Einfügen muss die ArrayList ggf. um Elemente erweitert werden, so dass sie groß genug ist.

Pseudocode:

```
einfuegen(n: int)
  k = n / 32
  solange Länge von daten ≤ k:
```

```
    hänge 0 an daten an
  bit = n % 32
  maske = 1 << bit
  daten[bit] = daten[bit] | maske
```

Bewertung

Vorteile:

- Sehr speichereffizient, wenn die Werte nicht zu groß sind
- Lesender Zugriff in konstanter Anzahl von Schritten möglich
- Schreibender Zugriff meistens schnell möglich

Nachteile:

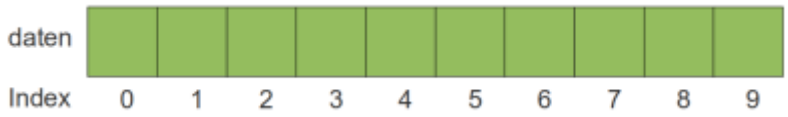
- Wenn nur wenige, aber große Werte gespeichert werden, wird Speicher verschwendet
- Konzept ist nur zum Speichern von Integer-Zahlen anwendbar

Variante 3: Hashtable

Die Hashtable verwendet ein Array, das die Werte der Menge enthält.

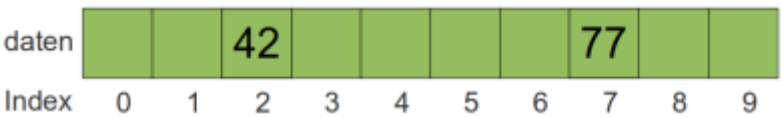
Das Problem ist, dass die Suche nach einem Wert unter Umständen sehr lange dauert – insbesondere, wenn es gar nicht in der Menge vorhanden ist.

Die Idee des Hashings ist, dass man einem Wert "ansieht", wo er im Array stehen muss. Wir nehmen als erstes Beispiel ein Array mit 10 Plätzen:



Um den Speicherort eines Wertes x zu bestimmen, berechnet man `h(x) = x mod arraylaenge` und legt ihn dort ab:

```
einfuegen(42) → Position 42 mod 10 = 2, daten[2] = 42
einfuegen(77) → Position 77 mod 10 = 7, daten[7] = 77
```



Um herauszufinden, ob ein Wert x in der Menge ist, berechnet man `h(x)` und prüft, ob der Wert dort

gleich x ist:

```
enthaelt(42) → Position 42 mod 10 = 2, daten[2] == 42 → ja
enthaelt(23) → Position 23 mod 10 = 3, daten[3] ist leer → nein
enthaelt(17) → Position 17 mod 10 = 7, daten[7] != 17 → nein
```

Dieses Vorgehen ist sehr schnell, da die Operationen nicht von der Anzahl der Werte in der Menge abhängig sind.

Die Funktion `h(x)`, die einem Element x seine Position im Array zuweist, wird als **Hashfunktion** bezeichnet.

Hashfunktionen bilden eine große Menge von Eingabewerten (hier: alle int-Zahlen) auf einen kleinen Bereich von Ausgabewerten ab (hier: die Zahlen von 0 bis 9).

Für unsere Datenstruktur ist die Hashfunktion

`h(x) = h mod arraylaenge`

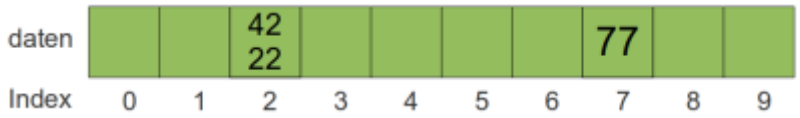
Kollisionen

"Aber was macht man, wenn zwei Zahlen den gleichen Hashwert erhalten?"

Wenn zwei unterschiedliche Zahlen den gleichen Hashwert bekommen, spricht man von einer **Kollision**. Kollisionen sind bei Hashfunktionen unvermeidlich, da die Menge der Eingabewerte um ein Vielfaches größer ist als die Menge der möglichen Ausgabewerte.

Im Beispiel würde die Zahl 22 ebenfalls am Index 2 abgelegt werden. Man benötigt also eine Strategie, wie man mit diesen Situationen umgeht.

Eine Möglichkeit ist, dass man an einer Stelle im Array nicht nur eine einzelne Zahl speichert, sondern mehrere (einen "Behälter", englisch "Bucket"):



Wenn man bestimmen will, ob ein Wert x vorhanden ist, geht man so vor:

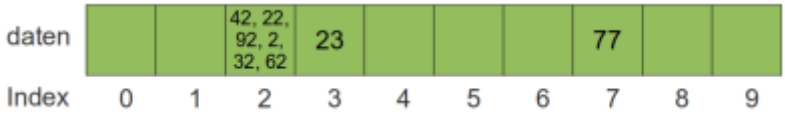
- Bestimme den Index von x (z.B. `h(22) = 2`)
- Untersuche alle Werte in diesem Behälter. Wenn einer davon gleich x ist, gib `true` zurück, sonst `false`

Ein Behälter kann z.B. mit einer `ArrayList` implementiert werden. Anstelle eines Arrays von int-Werten benötigt man jetzt also ein Array von `ArrayList<Integer>`-Objekten.

Der Speicheraufwand ist etwas höher und auch das Durchsuchen der Behälter dauert etwas länger.

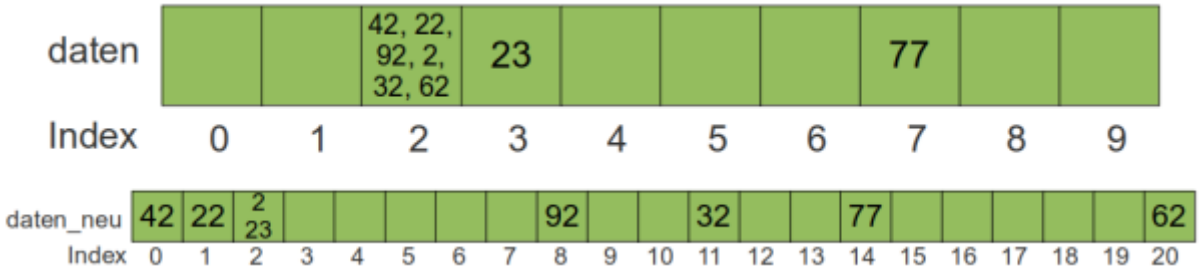
"Aber was haben wir damit jetzt gewonnen? Wir müssen immer noch die Behälter komplett durchsuchen!"

Das stimmt, allerdings ist es sehr unwahrscheinlich, dass alle Werte im gleichen Behälter landen. Wahrscheinlicher ist eine grobe Gleichverteilung.



Zudem kann man mit **Rehashing** die Elemente neu verteilen, wenn die Auslastung einen bestimmten Grenzwert (z.B. 75%) überschreitet.

Die Auslastung bezeichnet das Verhältnis der gespeicherten Werte zur Länge des Arrays.
Im Beispiel: 8 Elemente, 10 Plätze → Auslastung = 80%



Man legt ein neues Array der Länge z.B. 21 an und sortiert die bestehenden Werte neu ein. Viele der Kollisionen treten jetzt nicht mehr auf, es können aber neue Kollisionen entstehen, die Behälter enthalten jetzt im Durchschnitt weniger als einen Wert.

Die Rehashing-Operation ist sehr zeitaufwendig, muss aber nur relativ selten ausgeführt werden. Im Durchschnitt ist der lesende und schreibende Zugriff auf die Werte in konstanter Zeit möglich, also unabhängig von der Anzahl der Elemente im Set.

Wenn man im Vorfeld bereits weiß, wie viele Elemente vermutlich gespeichert werden sollen, kann man bereits beim Erzeugen das Array entsprechend anlegen.

Bewertung

Vorteile:

- Im Durchschnitt sehr schnell
- Auf beliebige Datentypen anwendbar (in Java: Methode `hashCode()` der `Object`-Klasse)

Nachteile:

- Hoher Speicherbedarf

- Kann in seltenen (konstruierten) Fällen langsam arbeiten