Teil 1: Brownies Collections: GapList und BigList

High-Performance Lists für Java

Jede Applikation benötigt Daten. Diese werden mit dem (Java) Collections Framework verwaltet, das deshalb zu den wichtigsten Teilen des JDK gehört. Es definiert Interfaces und stellt Klassen mit Standardimplementierungen bereit. Gerade für das am häufigsten verwendete Interface *java.util.List* fehlt aber eine problemlos einsetzbare Implementierung, denn auch die typischerweise genutzte Klasse *ArrayList* hat bei gewissen Operationen Schwächen. Die Brownies Collections Library stellt deshalb Alternativen bereit, die in allen Fällen schnell sind und gut skalieren.

von Thomas Mauch

Die Brownies Collections Library [1] besteht aus zwei Hauptteilen. In diesem ersten Teil der zweiteiligen Artikelserie stellen wir *GapList* und *BigList* mit zugehörigen Klassen vor, die das List-Interface implementieren. Ein zweiter, separater Artikel stellt dann die Key Collections vor, die es erlauben, Collections mit konfigurierbarer Funktionalität zur Laufzeit zu erzeugen.

Um das Problem mit der vom JDK zur Verfügung gestellten List-Implementierung zu illustrieren, sehen wir uns folgendes Beispiel an: Wir schreiben eine Applikation, die ankommende Ereignisse analysieren soll. Die Analyse geschieht immer in einem Fenster von fixer Größe, das die zuletzt angekommenen Elemente enthält. Um dieses Fenster zu unterhalten, brauchen wir also eine Liste, in der wir am Ende neue Elemente hinzufügen und am Anfang nicht mehr benötigte Elemente wieder löschen können. Die eigentliche Analyse wird dann mit Funktionen durchgeführt, die Zugriff auf die Elemente

Artikelserie

Teil 1: Brownies Collections: GapList und BigList
Teil 2: Key Collections

per Index benötigen. Das JDK bietet uns folgende zwei Klassen, die das List-Interface implementieren:

- *ArrayList* ist ideal für die Analysefunktionen, aber das Löschen von Elementen am Anfang der Liste ist – wie wir gleich sehen werden – problematisch.
- Bei *LinkedList* ist es genau entgegengesetzt: schnell beim Hinzufügen und Löschen von Elementen, aber nicht geeignet für effizienten Zugriff per Index.

Welche Implementierung man nun wählen soll, ist für einen Entwickler keine einfache Entscheidung. Schließlich weiß man, dass die gerade entwickelte Lösung nicht in jedem Fall performant sein wird – insbesondere, wenn sich auch die Größe des Analysefensters ändern kann. Um das schlechte Gewissen zu beruhigen, macht man vielleicht Performancemessungen, schreibt ein paar Worte in das Designdokument oder wenigstens einen Javadoc-Kommentar, damit die Kollegen auch sehen, dass zumindest die Problematik erkannt worden ist. Vielleicht fragt man sich aber auch, ob es denn keine Implementierung gibt, die beide Anforderungen erfüllt. Genau diese Lücke füllt *GapList*.

ArrayList

Was sind die Probleme, die beim Einsatz von *ArrayList* entstehen? Wie bereits in den Java-Tutorials erklärt wird, wird typischerweise die Klasse *ArrayList* als generell einsetzbare Implementierung verwendet, da sie meistens schneller ist und weniger Speicher braucht als *LinkedList* [2].

Das Tutorial erwähnt aber auch, dass Operationen, die nicht am Ende der Liste durchgeführt werden, langsam sein können. Dieses Performanceproblem hat seine Ursache in der Art der Datenspeicherung: *ArrayList* speichert die Elemente in einem Java-Array, wobei das erste Element immer an Position 0 gespeichert ist. Damit nun nicht bei jeder *add*-Operation ein neues Array alloziert werden muss, wird es immer schrittweise vergrößert, und die (noch) nicht genutzten Slots am Ende der Liste bleiben leer.

In **Abbildung 1** kann man einfach erkennen, weshalb das Einfügen am Ende schnell, am Anfang aber langsam ist: Am Ende kann direkt das neue Element gespeichert werden, während am Anfang oder in der Mitte zuerst existierende Elemente verschoben werden müssen, um Platz für das neue Element zu schaffen.

GapList

GapList optimiert die Datenspeicherung, sodass Operationen am Anfang und Ende der Liste immer schnell sind und an allen anderen Positionen durch Ausnutzen der Lokalitätseigenschaft schnell werden, sobald mehrere Operationen nacheinander in der Nähe stattfinden. Lokalitätseigenschaft (englisch: *Locality of Reference*) bedeutet, dass in einem gewissen Zeitabschnitt nur auf einen relativ kleinen Bereich der gesamten Datenmenge zugegriffen wird [3].

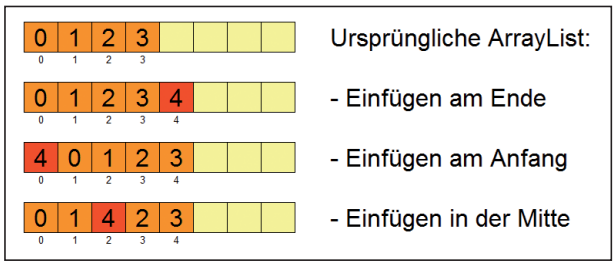

Abb. 1: Datenspeicherung in ArrayList

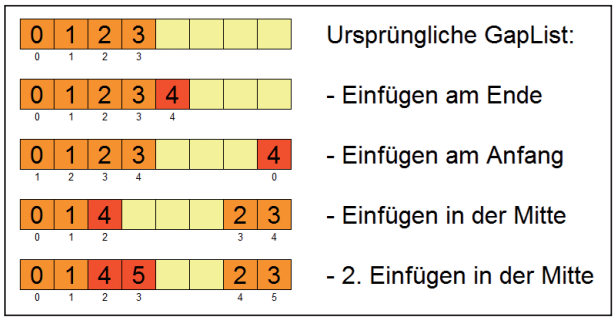

Abb. 2: Datenspeicherung in GapList

Um schnelle Operationen an Anfang und Ende der Liste zu erhalten, gibt es eine einfache Lösung: Anstatt dass wir beim Einfügen am Anfang alle Elemente verschieben, um Platz an Position 0 zu erhalten, belassen wir die Elemente an ihrem Platz und speichern das neue Element am Ende des Arrays. Wir nutzen das Array also als zyklischen Puffer. Für den Zugriff auf die Elemente speichern wir den Offset des ersten Elements im Array und berechnen damit die richtige Position.

Um die Lokalitätseigenschaft der Operationen ausnutzen zu können, erlauben wir ebenfalls eine Lücke (engl. *Gap*) im Array, das die Elemente speichert. Diese Gap der nicht benutzten Slots im Array kann an einem beliebigen Ort sein. Sie wird bei Bedarf automatisch erstellt, verschoben oder wieder gelöscht. **Abbildung 2** zeigt, wie die Elemente in *GapList* gespeichert werden.

Wie man sieht, müssen für das Einfügen am Anfang nun keine Elemente mehr verschoben werden, sodass diese Operation jetzt genauso schnell wie das Einfügen am Ende ist. Wenn Elemente in der Mitte eingefügt werden, müssen nur beim ersten Mal Elemente verschoben werden, um die Gap richtig zu platzieren. Bei der zweiten Einfügeoperation an derselben Position ist dies nicht mehr nötig. Damit ist die Operation entsprechend schnell. Dieses Ausnutzen der Lokalitätseigenschaft verbessert die Performance auch, wenn nachfolgende Operationen nicht an der exakt gleichen Position, sondern bloß in der Nähe stattfinden, da dann nur wenige Elemente verschoben werden müssen.

Das Entfernen von Elementen wird analog zum Einfügen behandelt und ist deshalb in *GapList* ebenfalls sehr effizient. Insbesondere garantiert *GapList*, dass das Entfernen am Anfang oder Ende der Liste immer ohne Verschieben von Elementen möglich und damit sehr schnell ist, während bei *ArrayList* auch das Entfernen von Elementen, die nicht am Ende sind, problematisch ist.

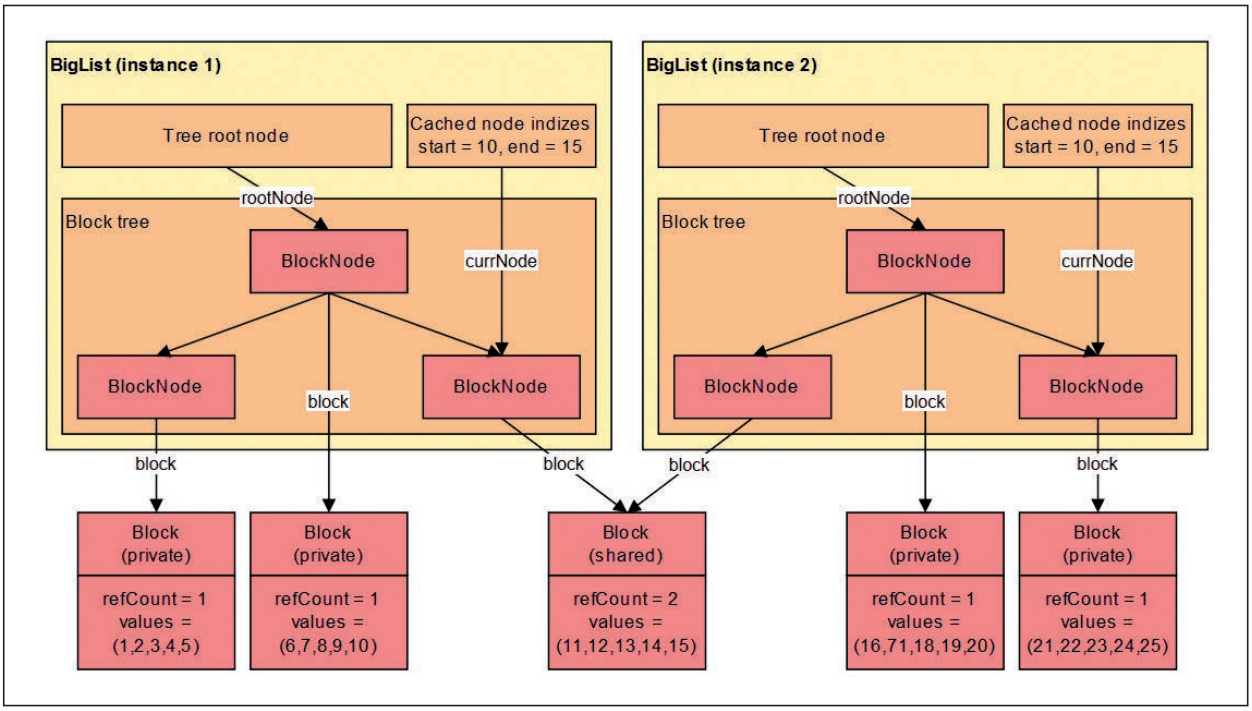

Abb. 3: Zwei BigList-Objekte teilen einen Datenblock

GapList vereint damit die Stärken von *ArrayList* und *LinkedList*, d. h. die Implementation ist schnell wie *ArrayList* beim Auslesen von Elementen, aber gleichzeitig auch schnell wie *LinkedList* beim Einfügen oder Entfernen an Anfang oder Ende. Zusätzlich profitieren alle Schreiboperationen von der Lokalitätseigenschaft der Zugriffe und sind damit typischerweise schneller als bei *ArrayList* oder *LinkedList*.

Damit *GapList* auch in bestehenden Anwendungen ohne großen Aufwand eingesetzt werden kann und die Migration möglichst einfach ist, wurde *GapList* als Drop-in-Ersatz für *ArrayList*, *LinkedList* und auch *ArrayDeque* konzipiert:

- *GapList* implementiert sowohl das Interface *java.util.List* als auch *java.util.Deque*.
- *GapList* implementiert alle *public*-Methoden von *ArrayList* (*ensureCapacity*, *trimToSize*).
- Wie *ArrayList* ist auch *GapList* nicht Thread-safe. Als einzige Abweichung bieten die Iteratoren keine Fail-fast-Funktionalität.

Mit der Implementierung des *Deque*-Interface bietet *GapList* auch bereits ein mächtigeres API als das spartanische, das in *ArrayList* zur Verfügung steht: Wie oft musste man schon umständlich *list.remove(list.size()-1)* schreiben, nur um das letzte Elements einer Liste zu löschen? Mit *java.util.Deque* ist dies mit *list.removeLast()* in einem Aufruf möglich. Zahlreiche weitere Komfortfunktionen wie *copy* werden über die Basisklasse *IList* zur Verfügung gestellt.

Ein weiterer Nachteil der JDK-Implementierungen ist die fehlende Unterstützung von primitiven Datentypen. Um die Daten eines primitiven Arrays wie *int[]* zu speichern, muss man deshalb eine *List<Integer>* verwenden, die ein Vielfaches an Speicher braucht. Die Brownies Collections Library bietet deshalb spezialisierte Implementierungen für alle primitive Datentypen. Es gibt pro Datentyp jeweils zwei verschiedene Klassen, z. B. für *int*:

- *IntGapList* arbeitet direkt mit dem Datentyp *int*, kann deshalb aber das Interface *java.util.List* nicht implementieren, das mit *Object* arbeitet.
- *IntObjGapList* nutzt intern eine *IntGapList* zur effizienten Datenspeicherung, implementiert aber *java.util.List*, sodass die Liste auch als *List<Integer>* verwendet werden kann. Die Elemente werden beim Zugriff durch Boxing/Unboxing automatisch konvertiert.

Analog zu *IList* gibt es auch für die spezialisierten Implementierungen eine Basisklasse. Für den Typ *int* heißt sie *IIntList*. Mit *IntGapList* kann damit nahezu identisch wie mit *GapList* gearbeitet werden. Einzig Iteratoren werden nicht unterstützt.

BigList

Leider löst aber *GapList* nicht alle Performanceprobleme. Bei *GapList* wie auch bei *ArrayList* kann bereits das Hinzufügen am Ende der Liste bei einer großen Zahl von gespeicherten Elementen langsam werden, wenn das allozierte Array voll ist: In diesem Fall muss ein neues, größeres Array alloziert werden, wohin dann alle Elemente kopiert werden müssen, was entsprechend speicher- und zeitintensiv ist.

Auch das Kopieren der ganzen Liste wird eine teure Operation, sobald die Anzahl gespeicherter Elemente groß genug ist. Kopieren ist aber eine häufige Operati-

on, z. B. wenn man über ein API eine Liste zurückgeben will, die unabhängig von der intern gespeicherten ist.

Eine Liste, die wirklich große Datenmengen speichern soll (wobei wir mit „groß“ meinen, dass sie noch immer in den Hauptspeicher passen), muss also spezielle Anforderungen erfüllen. Tut sie dies nicht, werden gewisse Operationen nicht nur langsam, sondern gar nicht mehr möglich sein, weil z. B. der zur Verfügung stehende Speicher nicht ausreicht:

- Der Speicher muss sparsam genutzt werden, sodass möglichst viele Elemente gespeichert werden können. Dies bedeutet beispielsweise, dass die Liste selbst wenig Speicher für eigene Metadaten brauchen oder dass es spezialisierte Versionen für primitive Datentypen geben soll.
- Alle Operationen müssen effizient sein, sowohl bezüglich der Performance als auch was den Speicherplatzbedarf betrifft. Die Kosten einer Operation sollen transparent sein, d. h. gleiche Operationen benötigen immer etwa gleich lange.
- Es muss möglich sein, viele Elemente auf einmal effizient verarbeiten zu können. Die Liste soll deshalb spezielle Methoden für Bulk-Operationen anbieten.

Die Klasse *BigList* ist unter Berücksichtigung dieser Punkte speziell für die Speicherung großer Datenmengen entwickelt worden. *BigList* speichert die Elemente in Blöcken fixer Größe. Eine Operation zum Einfügen oder Entfernen arbeitet deshalb nur mit den Elementen eines Blocks. Dadurch müssen wenig Daten kopiert werden, und die Operation ist entsprechend schnell.

Zusätzlich wird auf jedem Block ein Reference Counter [4] geführt, der die Verwendung eines Copy-on-Write-Ansatzes [5] erlaubt. Damit sind effiziente Kopieroperationen von großen Listen möglich, da die Elemente selbst nicht kopiert werden müssen. Die Blöcke werden in einer spezialisierten Baumstruktur verwaltet, und Informationen zum zuletzt verwendeten Block werden zur Ausnutzung der Lokalitätseigenschaft gecacht.

Ebenfalls bietet *BigList* spezielle Transfermethoden wie *transferCopy*, die das Kopieren von Daten direkt von Liste zu Liste erlauben, ohne dass die Elemente temporär in einer zusätzlichen Datenstruktur materialisiert werden müssen.

BigList: Details

Die Aufteilung der Elemente in die Blöcke erfolgt anhand ihres Indexes, der als Key fungiert. Für jede Operation auf einer *BigList* muss dann zuerst der Block bestimmt werden, in dem die betroffenen Elemente abgelegt sind. Um die Lokalitätseigenschaft ausnutzen zu können, speichert *BigList* bei jedem Zugriff den ermittelten Block. Wenn die nächste Operation dann denselben Block benötigt, kann dieser direkt wieder verwendet werden, ohne dass er erneut im Baum gesucht werden muss.

Die Defaultgröße der Blöcke beträgt 1 000 Elemente. Dieser Wert mag auf den ersten Blick klein erscheinen, er reicht aber aus, um bei Zugriffen mit Lokalitätseigenschaft ein effizientes Caching zu erlauben, und er garantiert, dass bei Schreiboperationen nur wenige Elemente herumkopiert werden müssen. Alternativ kann die Blockgröße auch beim Erstellen der *BigList* angegeben und damit den Anforderungen angepasst werden. Der erste Block wächst jeweils normal bis zur spezifizierten Größe. Alle anderen Blöcke werden bereits mit der fixen Blockgröße alloziert und wachsen oder schrumpfen nicht. Dadurch werden nicht unnötig Objekte alloziert und kopiert, und die Garbage Collection wird nicht belastet. Für die Speicherung der Elemente selbst wird natürlich eine *GapList* verwendet.

Die Defaultgröße der Blöcke beträgt 1 000 Elemente. Das reicht aus, um bei Zugriffen Lokalitätseigenschaften und effizientes Caching zu erlauben.

Wenn in einem Block, der schon voll ist, Elemente hinzugefügt werden müssen, wird er in zwei Blöcke gesplittet, um Platz zu schaffen. Wenn Elemente am Anfang oder Ende hinzugefügt werden, wird der Block nicht ganz, sondern nur zu 95 Prozent gefüllt. Dies erlaubt ein nachträgliches Einfügen von Elementen, ohne dass bei jeder Operation ein relativ kostspieliger Split notwendig ist.

Um Speicherplatz zu sparen, werden Blöcke auch wieder gemergt. Dies passiert automatisch, wenn nach einer Löschoperation zwei benachbarte Blöcke zu weniger als 35 Prozent gefüllt sind.

Um effizientes Kopieren mit einem Copy-on-Write-Ansatz zu ermöglichen, speichert *BigList* einen Reference Counter auf jedem Block, der angibt, ob der Block privat oder „shared“ ist. Am Anfang sind alle Blöcke privat mit einem Reference Count von 0, der Modifikationen erlaubt. Wenn eine Liste kopiert wird, wird der Reference Count für alle Blöcke um eins erhöht, die Blöcke selbst müssen aber nicht kopiert werden. Die Blöcke sind dadurch als „shared“ markiert, und Modifikationen sind verboten, während Lesezugriffe möglich sind. Bevor eine *BigList* eine Modifikation auf einem „shared“ Block machen kann, muss dann eine Kopie des Blocks gemacht werden. Dazu wird ein neuer Block mit allen Elementen des alten Blocks und Reference Count 0 erstellt, während der Reference Counter des alten Blocks um 1 erniedrigt wird. Der Reference Counter eines nicht mehr benutzten Blocks wird dann spätestens durch den Finalizer der *BigList* vor der Garbage Collection wieder erniedrigt. Dadurch ist sichergestellt, dass auch einmal geteilte Blöcke wieder freigegeben werden, sobald der Speicherplatz knapp

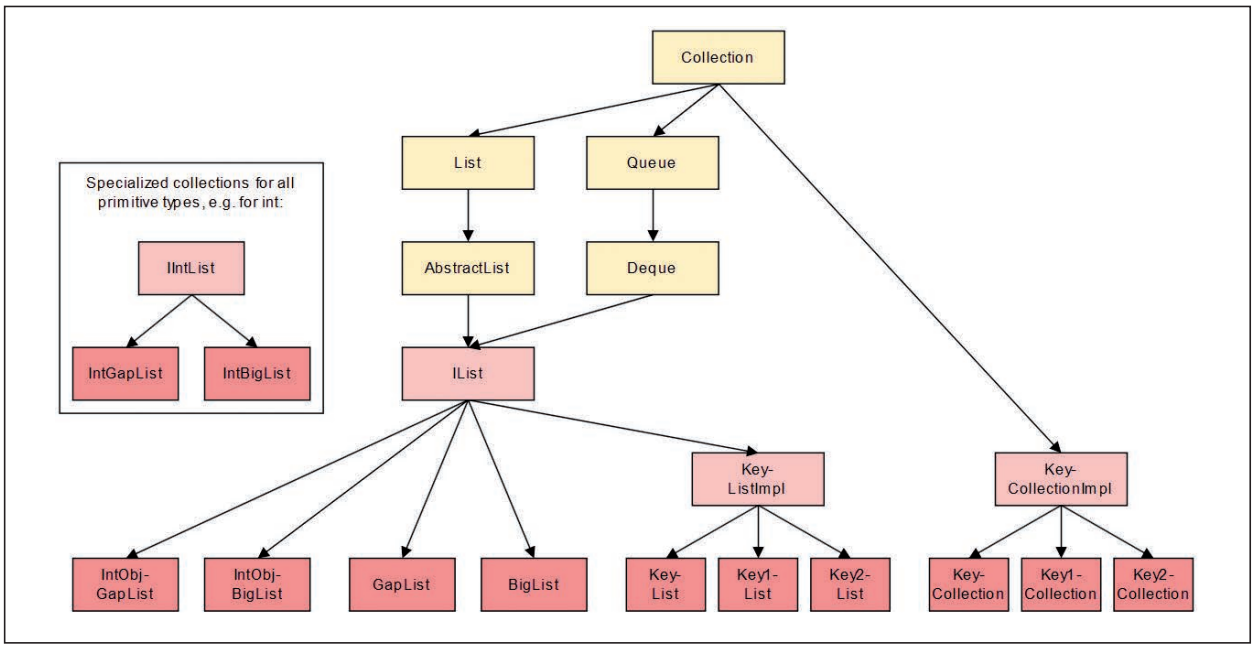

Abb. 4: Klassendiagramm

wird. Wenn der Freigabevorgang kontrolliert unter Programmkontrolle passieren soll, kann die Methode *clear* verwendet werden. **Abbildung 3** zeigt die Details der Implementierung für zwei BigList-Instanzen, die einen Block teilen.

Benchmarks

Für *GapList* und *BigList* wurden umfangreiche Benchmarks [6], [7] durchgeführt, um sicherzustellen, dass die erwartete Performance erreicht wird. Für diese Benchmarks wurde das Feld der Mitbewerber um zwei zusätzliche Kandidaten erweitert:

TreeList ist Teil von Commons Collections [8] und speichert Elemente in einer Baumstruktur. Wegen der Baumstruktur gibt es einen Speicher-Overhead für jedes einzelne Element. Es gibt keine wirklich langsamen Operationen, allerdings sind alle Operationen auch nicht wirklich schnell. Dies betrifft insbesondere auch Lesezugriffe.

FastTable ist Teil von javolution [9] und speichert Elemente in einer Fraktal-ähnlichen Datenstruktur. *FastTable* braucht wenig Speicher und ist grundsätzlich schnell. Allerdings müssen auch hier beim Einfügen und Löschen relativ viele Elemente verschoben werden. Insbesondere können nicht mehrere Elemente auf einmal effizient gelöscht werden, und die Datenstruktur schrumpft bei Löschungen nie, sodass einmal allozierter Speicher nicht für andere Zwecke verwendet werden kann.

Um die Performance zu bestimmen, wurde für alle sechs Kandidaten ein Benchmark mit neun Tests ausgeführt. Die Operationen wurden dabei jeweils ganz

Test	BigList	GapList	ArrayList	LinkedList	TreeList	FastTable
Get random	8.8	1.1	1.0	23'912.0	21.4	2.5
Add random	1.0	402.0	1'066.0	543.0	1.7	4.1
Remove random	1.0	257.0	300.0	1'176.0	4.3	15.2
Get local	1.9	1.3	1.0	357.0	6.5	1.9
Add local	1.0	16.5	1'256.0	9'382.0	6.2	31.3
Remove local	1.0	1.4	2'945.0	29'455.0	15.2	92.0
Add multiple	1.0	3.3	61.8	6.8	10.3	79.5
Remove multiple	1.0	22.0	4.8	59.7	792.0	7'097.0
Copy list	1.0	2.4	3.1	426.0	276.0	97.0

Tabelle 1: Performance der List-Implementationen

	int[]	IntBigList/ IntObjBigList	BigList	GapList	ArrayList	LinkedList	TreeList	FastTable
32 Bit	4 MB	4,2 MB	16,3 MB	16,0 MB	16,3 MB	20,0 MB	30,0 MB	16,1 MB
64 Bit	4 MB	4,3 MB	28,5 MB	28,0 MB	28,5 MB	36,0 MB	46,0 MB	28,2 MB

Tabelle 2: Speicherbedarf von Listen für 1 000 000 „int“-Elemente

zufällig mit einer gewissen Lokalität oder mit mehreren Elementen auf einmal ausgeführt. Tabelle 1 zeigt die Resultate für eine Liste von 1 000 000 Elementen.

Pro Test hat die jeweils schnellste Implementierung einen Wert von 1. Bei den anderen Implementierungen gibt der relative Performancefaktor an, wie viel langsamer diese waren. Hat also eine Implementation einen Wert von 3.0, ist sie für diesen Test drei Mal langsamer als die schnellste.

Um einen schnellen Überblick über die Resultate zu erhalten, sind die Resultatwerte entsprechend der Performance eingefärbt: Der beste Wert 1.0 ist immer grün, gute Werte (< 5.0) sind blau, akzeptable Werte (< 25.0) sind gelb und die restlichen, ungenügenden (>= 25.0) sind rot.

Die Tabelle zeigt, dass *BigList* mit Ausnahme von zwei Tests immer die schnellste Implementierung ist. Das einzige mittelmäßige Resultat stammt vom Test, wo absolut zufällig auf die Elemente zugegriffen wird. Es lässt sich damit erklären, dass es dann eben keine Lokalitätseigenschaft gibt, die ausgenutzt werden kann, sodass für jeden Zugriff der richtige Block in der Baumstruktur gesucht werden muss. Wie der Test *Get local* zeigt, ist *BigList* sofort wieder unter den schnellsten Implementierungen, sobald es eine gewisse Lokalität in den Zugriffen gibt, z. B. beim Iterieren über die Liste.

Ab welcher Größe lohnt sich der Einsatz von *BigList*? Leider lässt sich diese Frage nicht generell beantworten, da neben der Größe verschiedenste Faktoren eine Rolle spielen: Anzahl der Zugriffe, Verhältnis Lese- zu Schreibzugriffe, Art der Schreibzugriffe, Lokalität der Zugriffe etc. Grundsätzlich bietet *BigList* aber für jede Größe und alle Operationen eine gute Performance – wenn wir also eine einzige universelle List-Implementation benennen wollen, kann es nur *BigList* sein. Wenn wir die Größe der Liste irgendwie beschränken können, wählen wir *GapList* für kleinere und *BigList* für größere Listen.

Wir wollen nun noch untersuchen, wie sich der Speicherbedarf der verschiedenen Implementationen verhält und wie viel Speicher sich mit den spezialisierten Versionen für primitive Datentypen einsparen lässt. Tabelle 2 zeigt den benötigen Speicherplatz für eine Million *int*-Werte. Während der Speicherbedarf von *BigList*, *GapList* und *FastTable* im Wesentlichen dem der *ArrayList* entspricht, braucht *LinkedList* bereits 25 Prozent mehr Platz und *TreeList* fast doppelt so viel. Viel Speicherplatz sparen kann man hingegen mit dem Einsatz einer spezialisierten Version für primitive Datentypen, die nur unbedeutend mehr Speicher als ein simples Java-Array braucht. In einer 32-Bit-Umgebung braucht *IntBigList* nur 25 Prozent des Speichers von *BigList*, in einer 64-Bit-Umgebung sogar nur 14 Prozent. Dieser Unterschied ist damit zu erklären, dass ein einfaches Objekt in einer 32-Bit-Umgebung 8 Byte, in einer 64-Bit-Umgebung aber bereits 16 Bytes benötigt, während ein *int*-Wert in beiden Umgebungen genau 4 Byte belegt.

Fazit und Ausblick

Wie wir gesehen haben, bietet das JDK gerade für das meist verwendete Collection-Interface *java.util.List* keine universell einsetzbare Implementierung. Die Brownies Collections Library füllt diese Lücke mit *GapList* und *BigList*.

GapList bietet eine großartige Performance für alle Operationen und kann als Drop-in-Ersatz für *ArrayList*, *LinkedList* und *ArrayDeque* verwendet werden. *BigList* skaliert zusätzlich auch für große Datenmengen und garantiert, dass alle Operationen schnell und mit geringem Speicherbedarf ausgeführt werden können. Für beide Klassen gibt es spezialisierte Varianten für primitive Datentypen.

Die Brownies Collections Library ermöglicht es damit dem Entwickler, mit geringem Programmieraufwand Applikationen zu erstellen, die effizient bezüglich Performance und Speicherplatzverbrauch sind und dadurch gut skalieren. **Abbildung 4** zeigt alle Klassen der Library im Überblick.

Die Abbildung enthält damit auch bereits die Key Collections, welche die einfache Definition von mächtigen Datenstrukturen erlauben, denen die gewünschte Funktionalität deklarativ zugewiesen werden kann. Diese werden in einem separaten Artikel in der nächsten Ausgabe vorgestellt.

Thomas Mauch arbeitet als Software Architect bei Swisslog in Buchs, Schweiz. Seine Schwerpunkte liegen im Bereich Java und Datenbanken. Er interessiert sich seit den Zeiten des C64 für alle Aspekte der Softwareentwicklung.

Links & Literatur

[1] http://www.magicwerk.org/collections

[2] http://docs.oracle.com/javase/tutorial/collections/implementations/list.html

[3] http://de.wikipedia.org/wiki/Lokalit%C3%A4tseigenschaft

[4] http://de.wikipedia.org/wiki/Referenzz%C3%A4hlung

[5] http://de.wikipedia.org/wiki/Copy-On-Write

[6] http://java.dzone.com/articles/gaplist-%E2%80%93-lightning-fast-list

[7] http://java.dzone.com/articles/biglist-scalable-high

[8] http://commons.apache.org/proper/commons-collections

[9] http://javolution.org/apidocs/javolution/util/FastTable.html