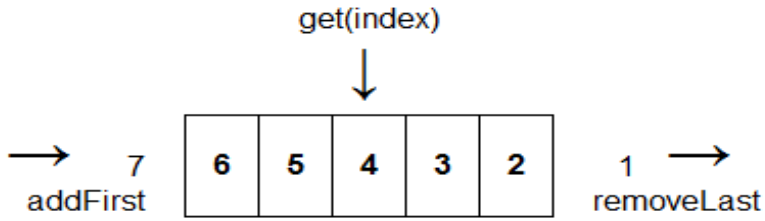
Thomas Mauch

High Performance Lists in Java

Motivation: Praxis-Beispiel

Applikation zur Datenanalyse:

- Speichert ein Fenster fixer Grösse geordnet nach Zeit



- Neue Ereignisse werden am Anfang hinzugefügt und alte am Ende gelöscht
- Verschiedene analytische Funktionen können auf die gespeicherten Elemente angewandt werden (benötigen Zugriff per Index)

JDK Collections

https://docs.oracle.com/javase/tutorial/collections/implementations/list.html

Most of the time, you'll probably use `ArrayList`, which offers constant-time positional access and **is just plain fast**...

If you frequently add elements to the beginning of the `List` or iterate over the `List` to delete elements from its interior, you **should consider** using `LinkedList`...

But you **pay a big price in performance**, positional access requires linear-time in a LinkedList...

Fazit

- ArrayList: gut für Analyse der Daten, schlecht für das Hinzufügen von Elementen am Anfang der Liste
- LinkedList: umgekehrt, gut für Hinzufügen / Entfernen, schlecht für Analyse

Was soll man als Entwickler tun?

- Vermutlich wird man ArrayList wählen (mit einem schlechten Gefühl)

ArrayList: Keine primitive Typen

- Wir müssen einfache int Werte speichern, aber ArrayList<Integer> speichert ein Integer-Object für jeden Wert

Wieviel Speicher wird gebraucht?

- 4 (!) mal mehr Speicher in einer 32 bit Umgebung
- Bis 7 (!!) mal mehr bei 64 bit (vor Java 8, ohne compressed OOPs)

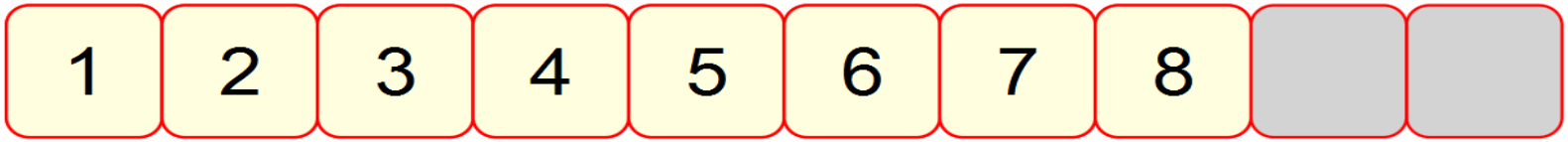
Performance: Unterschied etwa 10%

ArrayList: Minimalistisches API

- Wie häufig mussten wir schon list.get(list.size()-1) schreiben anstatt getLast()?
- Nicht nur ein kosmetisches Problem für Collections.synchronizedList()
- ArrayList implementiert nur List...
- LinkedList implementiert List und Deque...
- ArrayDeque implementiert nur Deque...

ArrayList: Implementierung

- ArrayList speichert die Elemente zusammenhängend in einem Array
- Wenn mehr Platz benötigt wird, wird das Array vergrössert
- Das Array schrumpft nicht automatisch



ArrayList: Methoden

- Animation

GapList: Ziele

- Performance-Optimierung durch Verbesserung des Speicherlayout von ArrayList
- Operationen am Anfang der Liste sollen so schnell sein wie am Ende
- Locality of Reference ausnutzen um Operationen zu beschleunigen, welche nahe beeinander statt finden

GapList: Zyklischer Buffer

- **Ziel**: Operationen am Anfang der Liste sollen so schnell wie am Ende sein
- **Lösung**: Array wird als zyklischer Buffer genutzt
- Der physische Arrayindex wird aus dem logischen mit einer Modulo-Operation berechnet

GapList: Gap

- **Ziel**: Locality of Reference ausnutzen um Operationen zu beschleunigen, welche nahe beeinander statt finden
- **Lösung**: Array kann einen Gap enthalten
- Der Gap wird bei Bedarf automatisch erstellt, verschoben und entfernt

GapList: Methoden

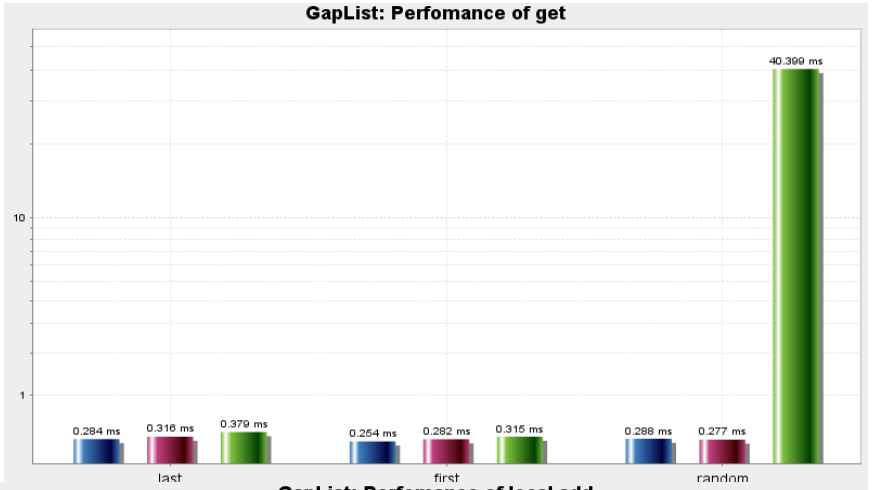
- Animation

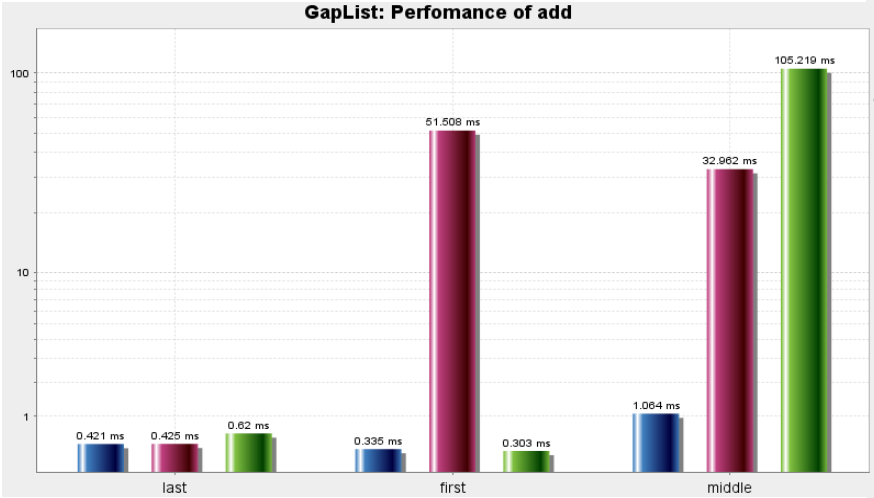
GapList: Performance

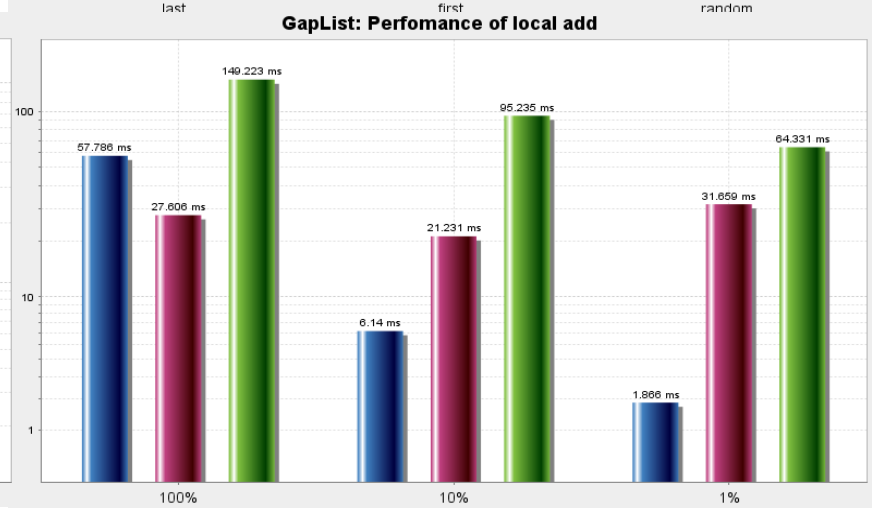
- Exzellente Performance
- Schneller Zugriff per Index (wie ArrayList)
- Schnelles Einfügen und Löschen an Anfang und Ende (wie LinkedList)
- Locality of Reference macht wiederholte Zugriffe auf andere Elemente schnell

GapList: Benchmarks

GapList ist schnell!
(Skala logarithmisch)







GapList: Features

- Interface IList extends List, Deque
- Support für primitive Typen:
  - IntGapList arbeitet mit int
    (Interface IIntList)
  - IntObjGapList arbeitet mit Integer
    (implementiert IList)

Empfehlung:

- GapList als Ersatz für
  ArrayList, LinkedList, ArrayDeque nutzen

GapList als universelle Liste?

- Es werden Elemente am Ende zu einer Liste hinzugefügt, bis sie sehr gross ist (10 Mio Elemente)
- Gemessene Zeit für einzelne add-Operation:
  - Meistens ok (min: 0 ms, avg: 0 ms)
  - Aber manchmal schlecht: 34 ms!
- Auch andere Operationen skalieren mit grossen Liste nicht (z.B. Kopieren)

Grosse Collections

Spezielle Anforderungen

- Memory muss sparsam genutzt werden (für Objekte und primitive Typen)
- Alle Operationen müssen effizient bezgl. Performance und Speicher sein (Kopieren)
- Unterstützung für Bulk-Operationen

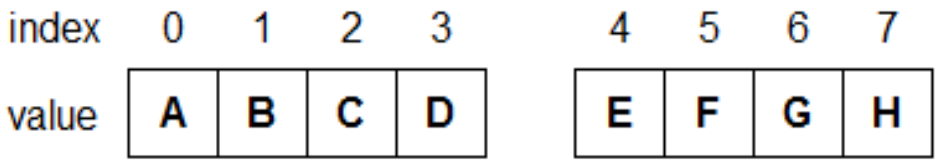
BigList wurde für diese Anforderungen entwickelt

BigList: Ziele

- **Ziel**: Beim Hinzufügen/Löschen sollen wenige Elemente bewegt werden
- **Lösung**: Elemente werden in Blöcken gespeichert
- **Ziel**: Das Kopieren von grossen Listen soll schnell sein
- **Lösung**: Blöcke können von Listen gemeinsam genutzt werden

BigList: Blöcke

- Elemente werden in Blöcken fixer Grösse gespeichert



- Wenn ein Block voll ist, werden die Elemente automatisch auf zwei Blöcke verteilt
- Zwei benachbarte Blöcke werden automatisch zusammengeführt, wenn sie weniger als ein Drittel gefüllt sind
- Blöcke sind als GapList implementiert

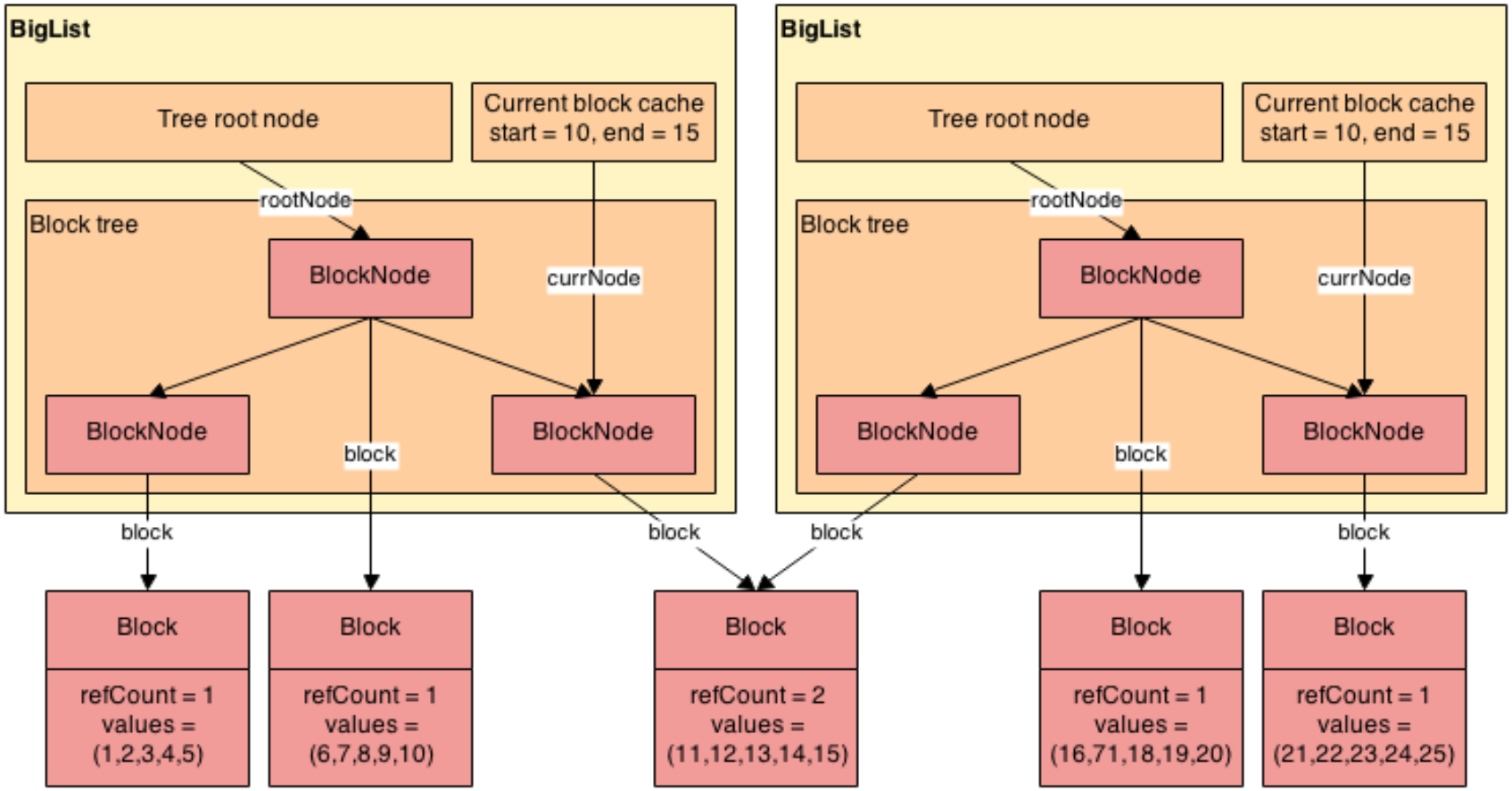
BigList: Verwalten von Blöcken

- Für jede Operation muss zuerst der betroffene Block bestimmt werden
- Die Blöcke werden deshalb in einer spezialisierten Tree-Struktur verwaltet
- Die Information für den zuletzt genutzten Block wird gecached, um die Locality of Reference ausnutzen zu können

BigList: Teilen von Blöcken

- Jeder Block speichert einen Reference Count um Copy-on-Write zu unterstützen
- Am Anfang haben alle Blöcke einen Reference Count von 1
  (Block privat, Modifikation möglich)
- Wenn eine Liste kopiert wird, muss nur der Reference Count erhöht werden
- Reference Counter wird durch clear() oder den Finalizer zurückgesetzt

BigList: Implementierung



BigList: Methoden

- Animation

BigList: Features

- BigList implementiert ebenfalls IList
- setAll(index, coll) / setArray(index, elems...) / setMult(index, len, elem)
- Internes Kopieren: copy, move, etc.
- Externes Kopieren: transferCopy, etc.
- Auch BigList hat Klassen für primitive Typen: IntBigList, IntObjBigList

BigList: Benchmarks

Liste mit 1'000'000 Elementen
Gewinner: BigList!

	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	1.0	2.4	3.1	426.0	276.0	97.0

Universelle Listen

Rezept:

- Benutze GapList anstelle von ArrayList / LinkedList / ArrayDeque
- Benutze BigList anstatt GapList, wenn die Liste grösser werden kann

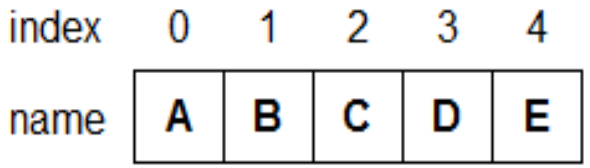
Wie immer: Am besten messen und nicht schätzen...

Teil 2:

KeyCollections

Motivation

Wir müssen eine Liste von Spalten mit Namen verwalten



- Spalten haben eine Ordnung
- Spaltennamen sind einzigartig
- Effizienter Zugriff per Index und Name

Benötigen wird List oder Map oder beides?

Wie lange braucht die Implementierung?

Lösung

- Key1List<Column,String> cols =
  new Key1List.Builder<Column,String>.
  withKey1Map(Column::getName).
  withPrimaryKey1().build();
- class Column {
  String name;
  String getName() {};
  }

Constraints

- class Table {
  List<Column> cols;
  List<Column> getCols() {
  return cols;
  }
  }
- Unsicher!

Constraints

- class Table {
  List<Column> cols;
  return Collections.unmodifiableList(cols);
  }
  }
- Nur Lese-Zugriff!

Constraints

- ```
  class Table {
      void addColumn(Column col) {
          check(col);
          cols.add(col);
      }
  }
  ```
- Beschränktes API!
  Oder mühsam zum Schreiben!
  ```

Constraints

- class Table {
  void setColumn(List<Column> cols) {
  check(cols);
  this.cols = new List<>(cols);
  }
  }
- Ineffizient!

Constraints

- class Table {
  Key1List<Column> cols...
  Key1List<Column,String> getCols() {
  return cols;
  }
  }
- Einfach!
- Constraints sind wichtig für ein mächtiges API

Keys

- Ein Key ist ein Wert, welcher mit einer Mapper-Funktion aus einem Element extrahiert wird
  - Key1List<Column,String> withKey1Map(Column::getName)
- Mit diesem Ansatz arbeiten alle Key Collections mit einer Collection, das separate Map Interface wird nicht gebraucht

Keys und Maps

- Ein typischer Fall:
  - class Column { String key; Column col; }
- Typischer Fall mit JDK:
  - Map<String,Column> cols;
    col = new Column("name");
    cols.put(col.getName(), col);
- Untypischen Fall mit Key Collections:
  - class KeyColumn { String key; Column col; }

Key Map und Element Set

- Wir nennen die Menge aller Key Werte, welche von einer Funktion extrahiert werden, KeyMap:
  - withKey1Map
- Auch die Elemente selbst können als Key benutzt werden:
  - withElemSet
- Keys ermöglichen schnellen Zugriff:
  - getByKey1("A")

Keys und Constraints

- Keys können benutzt werden, um Constraints zu definieren:
  - Nullwerte: erlaubt / verboten
    withKey1Null
  - Duplikate: erlaubt / verboten
    withKey1Duplicates
- Shortcuts:
  - withPrimaryKey1: keine Nullen / Duplikate
  - withUniqueKey1: Nur Null-Duplikate

Keys und Sortierung

- Keys können ebenfalls zur Sortierung verwendet werden
- Die Reihenfolge der Keys kann also zufällig oder geordnet sein (mit natürlichem oder spezifischen Comparator)
- Wenn ein Key sortiert ist, ist die Collection nicht automatisch auch sortiert
- Die Option withOrderBy ermöglicht dies

Mehr Constraints

- Nur positiven Zahlen
  withConstraint(i -> i>=0)
- Maximale Grösse der Collection
  withMaxSize(10)
- Maximale Grösse als Fenster
  withWindowSize(10)
- Triggers
  withBeforeInsert / withBeforeDelete
  withAfterInsert / withAfterDelete

Constraints im Alltag

- Constraint List:

```
list = new KeyList.Builder<String>().
      withPrimaryElem().build();
list.addArray("a", "b");
```

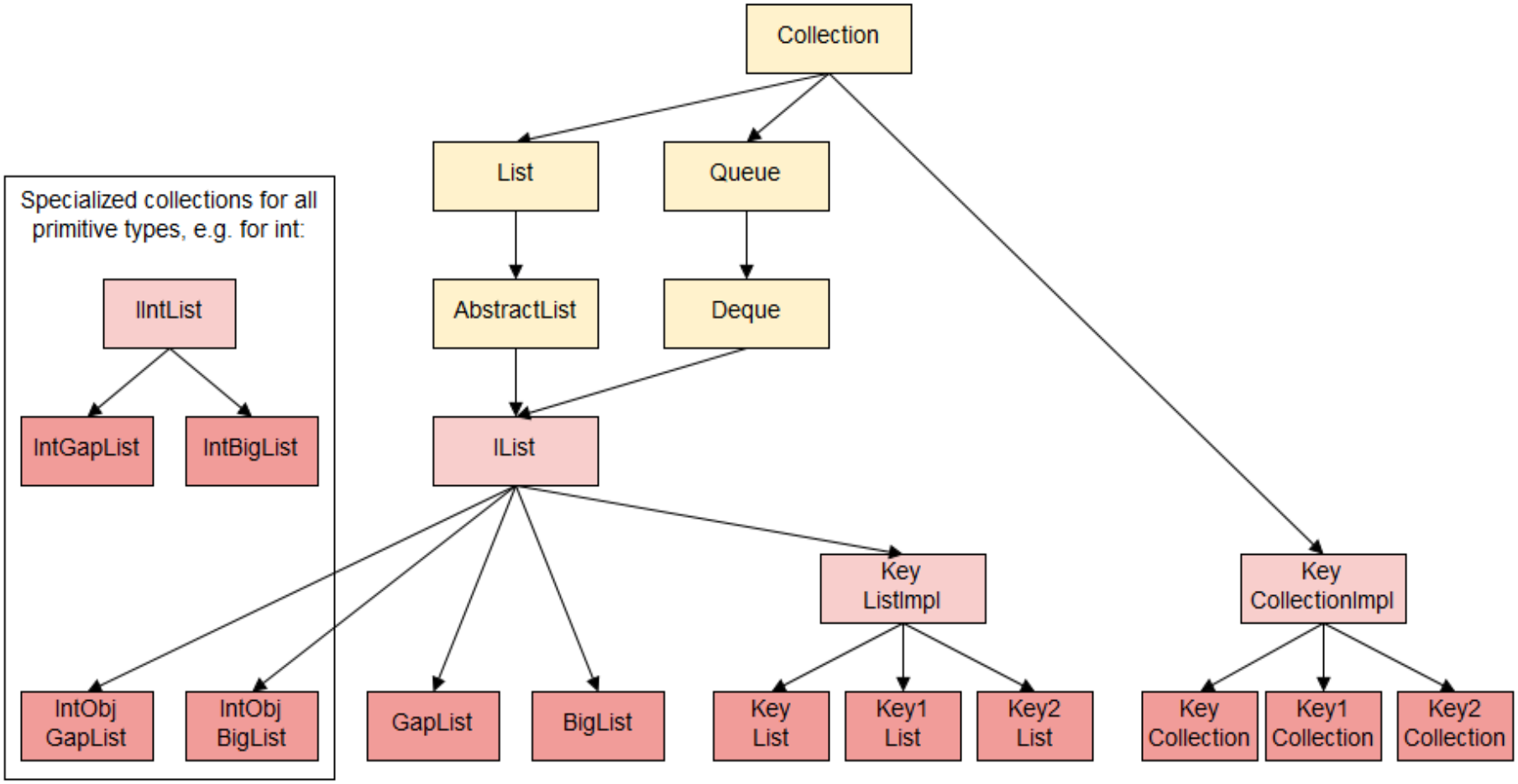
- Einzel-Operation verletzen Constraint:

```
list.set(0, "b"); list.set(1, "a"); // ERROR
```

- Mächtigere Operationen benutzen:

```
list.setArray(0, "b", "a");
list.reverse(0, 2); list.swap(0, 1, 1);
```

Klassen


Collection
List
Queue
AbstractList
Deque
IList
Specialized collections for all primitive types, e.g. for int:
IIntList
IntGapList
IntBigList
IntObj GapList
IntObj BigList
GapList
BigList
Key ListImpl
Key List
Key1 List
Key2 List
Key CollectionImpl
Key Collection
Key1 Collection
Key2 Collection

API Übersicht

- KeyList Klassen implementieren das IList Interface (also List und Deque)
- KeyCollection Klassen implementieren das JDK Collection Interface
- Mit asSet()/asMap() kann als JDK Set oder Map auf die Elemente zugegriffen werden

API Methoden

- Element Set
  contains, remove, indexOf,
  put, getAll, getCount, removeAll
- Key Maps
  containsKey1, getByKey1, getAllByKey1,
  getCountByKey1, getDistinctKeys1,
  putByKey1, removeByKey1,
  removeAllByKey1, indexOfKey1

API Beispiel

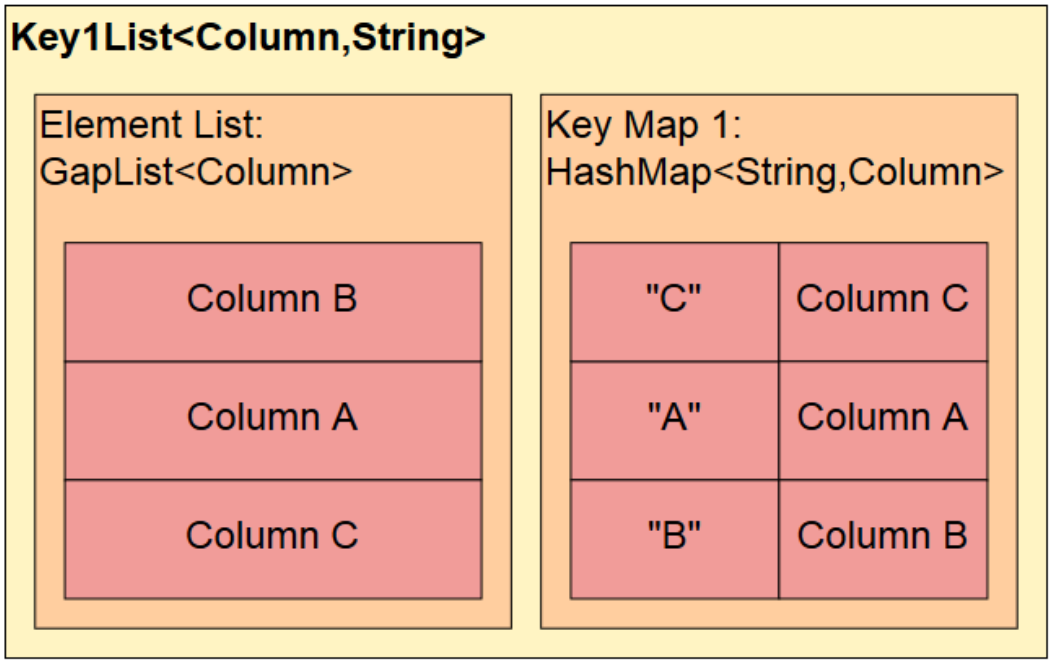
- cols.add(new Column("A"));
  - ok
- Column col = cols.getByKey1("A");
  - retrieve element
- cols.add(new Column("A"));
  - fails with DuplicateKeyException
- cols.putByKey1(new Column("A"))
  - replaces element
- cols.removeByKey1("A")
  - removes element

Implementierung

- Key Collections nutzen andere Collections als Building Blocks:
  - GapList / BigList
  - HashMap / TreeMap
- Die benötigten Komponenten werden anhand der Konfiguration bestimmt, wenn die Collection erstellt wird

Layout

Komponenten: GapList / HashMap



Performance

KeyList mit Element Set:

- contains: schnell
- containsAll: schnell (basiert auf contains)
- remove(int): schnell
- remove(Object): langsam (Iterieren in Liste)

Sortierte KeyList mit Element Set:

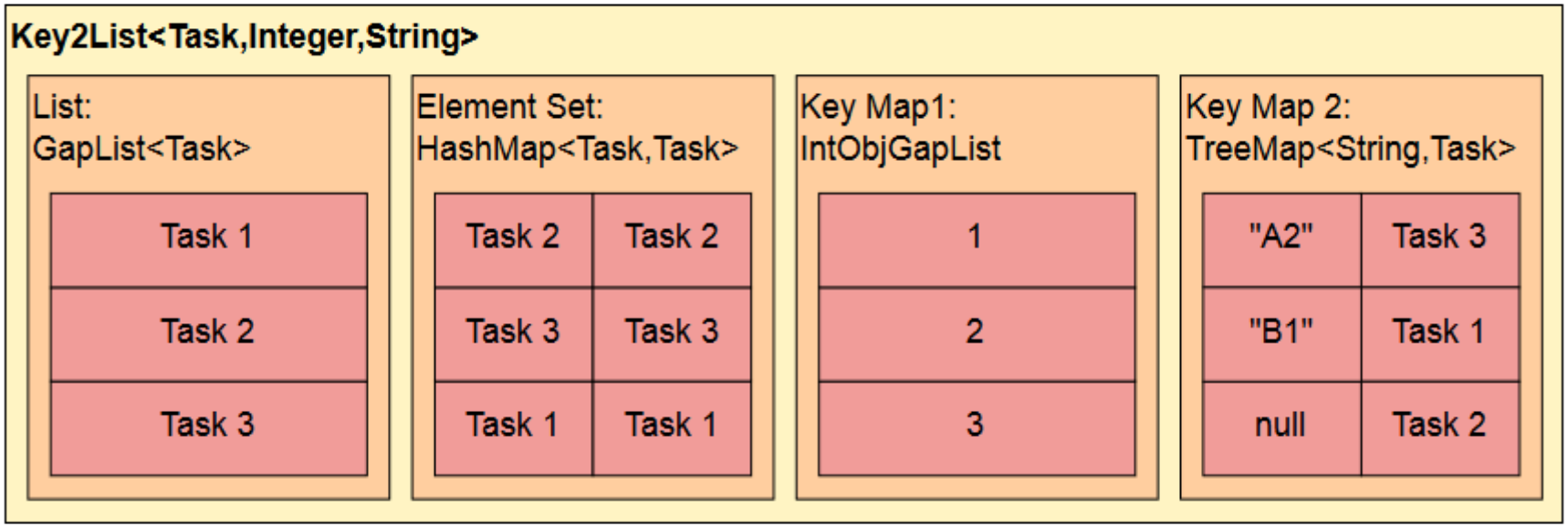
- remove(Object): schnell

Beispiel UseItAll

- Key2List<Task,Integer,String> list =
  new Key2List.Builder<Task,Integer,String>().
  withElemSet().
  withKey1Map(Task::getId).withPrimaryKey1().
  withOrderByKey1(int.class).
  withKey2Map(Task::getExtId).withUniqueKey2().
  withKey2Sort(true).
  build();
- class Task {
  int id; String extId;
  }

Layout UseItAll

Komponenten:
GapList / HashMap / IntObjGapList / TreeMap



Typ Parameter im Alltag

- Typ Parameter sind mühsam zum Tippen und Lesen
- Key1List<Column,String> cols =
  new Key1List.Builder<Column,String>.
  withKey1Map(Column::getName).
  withPrimaryKey1().build();
- void func(Key1List<Column,String> cols) {}

Typ Parameter vereinfacht

- Type Parameter sind verborgen
- class ColumnList extends
  Key1List<Column,String> {
  ColumnList() {
  getBuilder().
  withKey1Map(Column::getName).
  withPrimaryKey1().build();
  }}
- void func(ColumnList cols) {}

Beispiel: Constraint List

- Eine Liste, welche nur positive Integer akzeptiert

```
new KeyList.Builder<Integer>().
    withConstraint(i -> i>=0).build()
```

- Eine Liste, welche nur 10 Elemente akzeptiert

```
new KeyList.Builder<Integer>().
    withMaxSize(10).build()
```

Beispiel: Set List

- SetList kombiniert eine Liste mit einem Set
- Eine Set List mit Null und Duplikaten

```
new KeyList.Builder<String>().
    withElemSet().build()
```

- Eine Set List mit einzigartigen Elementen

```
new KeyList.Builder<String>().
    withPrimaryElem().build()
```

- Das Element Set wird automatisch erstellt

Beispiel: Sorted List

- Eine Liste von Files nach Namen geordnet
- Key1List<File,String> list =
  new Key1List.Builder<File,String>().
  withKey1Map(File::getName).
  withPrimaryKey1().withKey1OrderBy(true).bui
  ld();
- Soll es eine sortierte Liste überhaupt geben?

Beispiel: Multi Set

MultiSet speichert nur die Anzahl von gespeicherten Duplikaten

- Unsortiertes MultiSet
  `new KeyCollection.Builder<String>.withElemCount().build()`
- Sortiertes MultiSet
  `withElemSort(true)`

Die Option `withElemCount()` gibt es nur für die Klasse KeyCollection

Beispiel: Bidirectional Map

- Eine BiMap mit sortierten Schlüsseln
- ```
  Key2Collection<Zip,Integer,String> zips =
  new Key2Collection.Builder<Zip,Integer,String>().
  withKey1Map(Zip::getCode).withPrimaryKey1().
        withKey1Sort(true).
  withKey2Map(Zip::getCity).withKey2Sort(true).
        build();
  ```
- ```
  class Zip {
        int code; String city;
  }
  ```

ENDE

Mehr Informationen:

- www.magicwerk.org