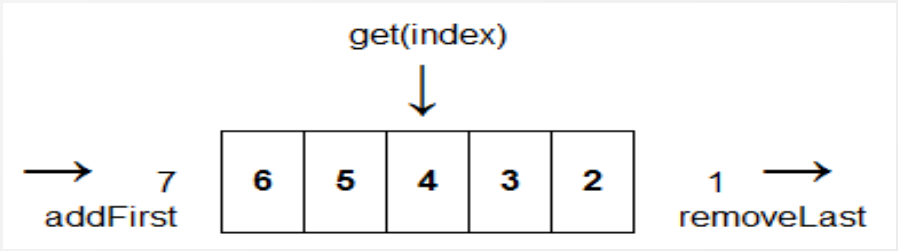
High Performance Lists in Java

Thomas Mauch
JavaZone 2015

Motivation: A Real World Example

The Analytical Engine:



- maintain a fixed window of events ordered by time
- new events are added at the head and disappear at the end
- various analytical functions can be applied to the events in the window

The 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...

Analyis

- ArrayList would excel in the analysis part, but behave poorly in adding the events
- LinkedList: vice versa

What will the good programmer do?

- Probably choose ArrayList with a bad feeling

ArrayList: Not For Primitive Types

We need to store just int values, but ArrayList<Integer> stores an Integer object for each value

How much memory is needed/wasted?

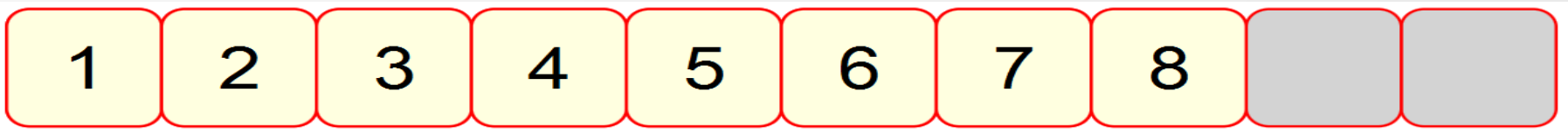
- 4 (!) times more memory needed
  in a 32 bit environment
- 7 (!!) times more memory needed
  in a 64 bit environment

ArrayList: Minimalistic API

- How many times did I have to write list.get(list.size()-1) instead of getLast?
- ArrayList only implements List...
- LinkedList implements List and Deque...
- And ArrayDeque only implements Deque...
- Collections.synchronizedList() is not enough

ArrayList: Implementation

- ArrayList stores the elements contiguously in an array
- If there is not enough space, the array is extended
- The array does not shrink automatically



ArrayList: Operations

- Animations

GapList: Goals

Improve storage layout used by ArrayList to

- Make operations at the head of the list as fast as at the tail
- Exploit locality of reference to speed up operations happening near to each other

Locality of reference?

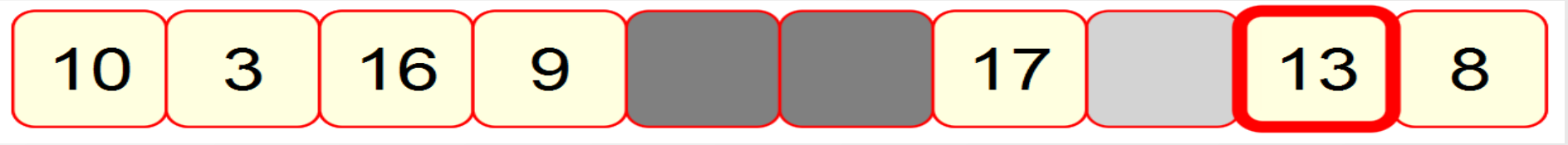
- Near storage locations are frequently accessed at the same time

GapList: Cyclic Buffer

- Goal: Make operations at the head of the list as fast as at the tail
- Solution: Use array as cyclic buffer
- Physical array index is created from the logical one using a modulo operation

GapList: Gap

- Goal: Exploit locality of reference to speed up operations happening near to each other
- Solution: Allow one gap within array
- The gap is automatically created, moved, and removed as needed



GapList: Operations

Animations

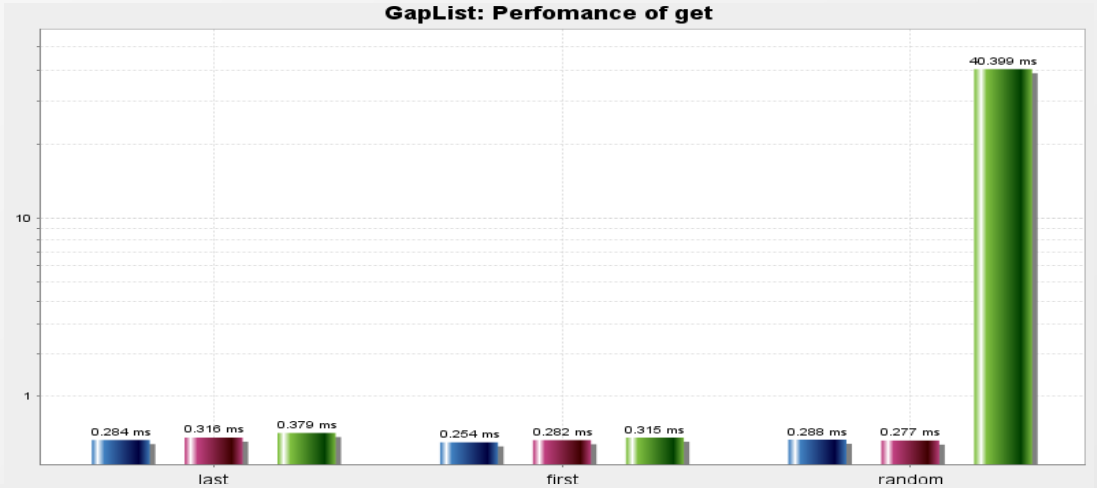
GapList: Performance

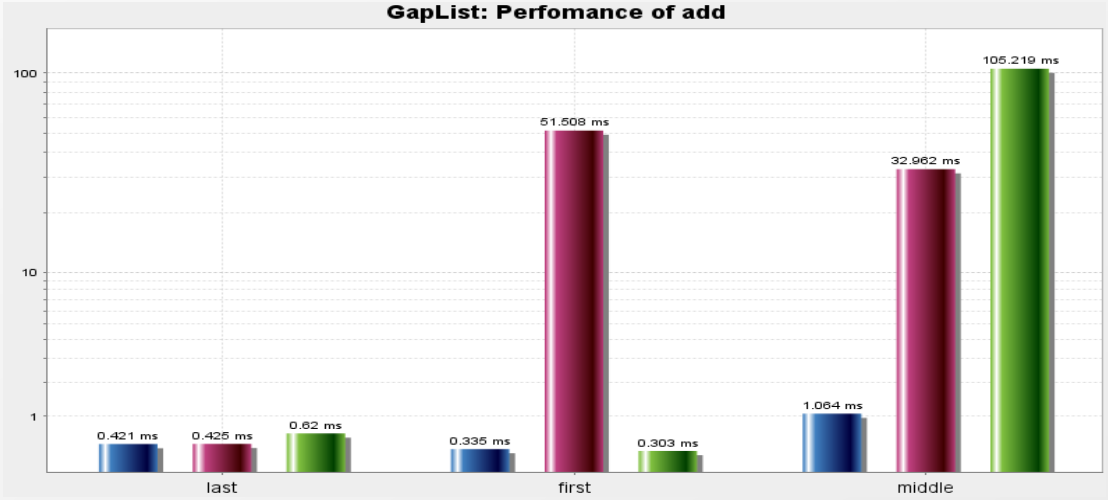
Great Performance

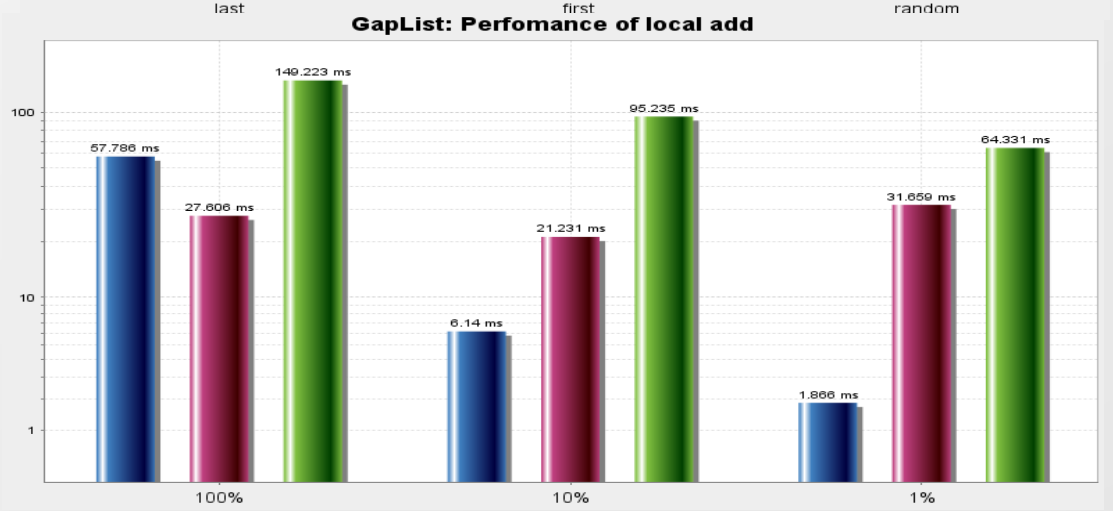
- Efficient access to elements by index (as ArrayList)
- Efficient insertion and deletion at head and tail of list (as LinkedList)
- Exploit the locality of reference for all other locations

GapList: Benchmarks

- Bechmarks prove GapList is fast!





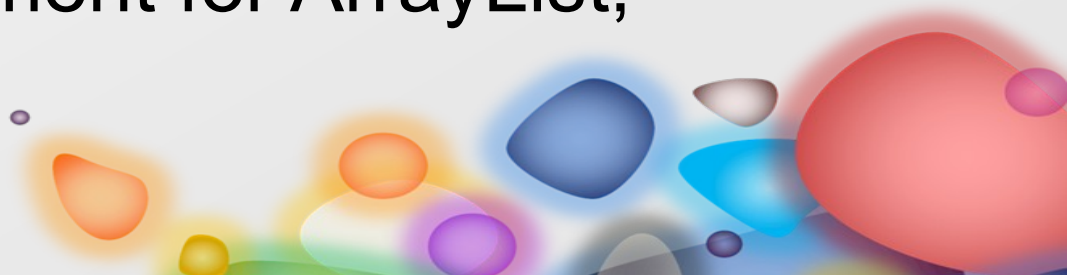


GapList: Features

- Interface IList extends List, Deque
- Support for storing primitive types:
  - IntGapList returns int (interface IIntList)
  - IntObjGapList returns Integer
    (and can therefore implement IList)

Recipe

- Use GapList as drop-in replacement for ArrayList, LinkedList, ArrayDeque



Is GapList the Universal List?

- Repeat adding elements to the end of a list until it becomes large (10 mio elements)
- Elapsed time in for a single add operation:
  - Most of the time ok (min: 0, avg: 0 ms)
  - But sometimes bad: max: 34 ms!
- Also other operations on large collection do not scale (e.g copying)

Large Collections

Special Requirements

- Memory must be used wisely
- All operations must be efficient regarding performance / memory (e.g. copying)
- Support for bulk operations

BigList has been designed to meet these requirements

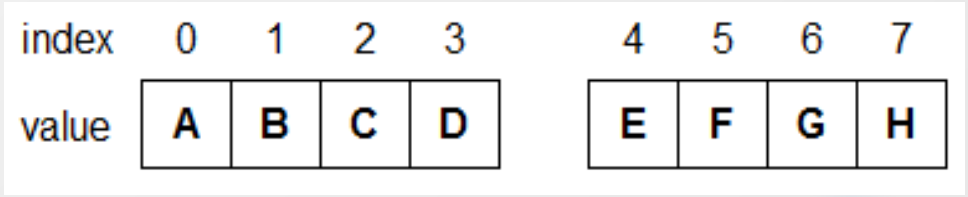
BigList: Goals

- Goal: Avoid moving many elements on insert / removal
- Solution: Store elements in small blocks

- Goal: Make copying large collections fast
- Solution: Share blocks between instances

BigList: Blocks

- Elements are stored in fixed-sized blocks (no copying needed, default size 1000)
- If a block is full, the content is automatically split in two blocks
- Blocks are also merged if two adjacent blocks are both filled less than 35%
- Blocks are stored as GapList



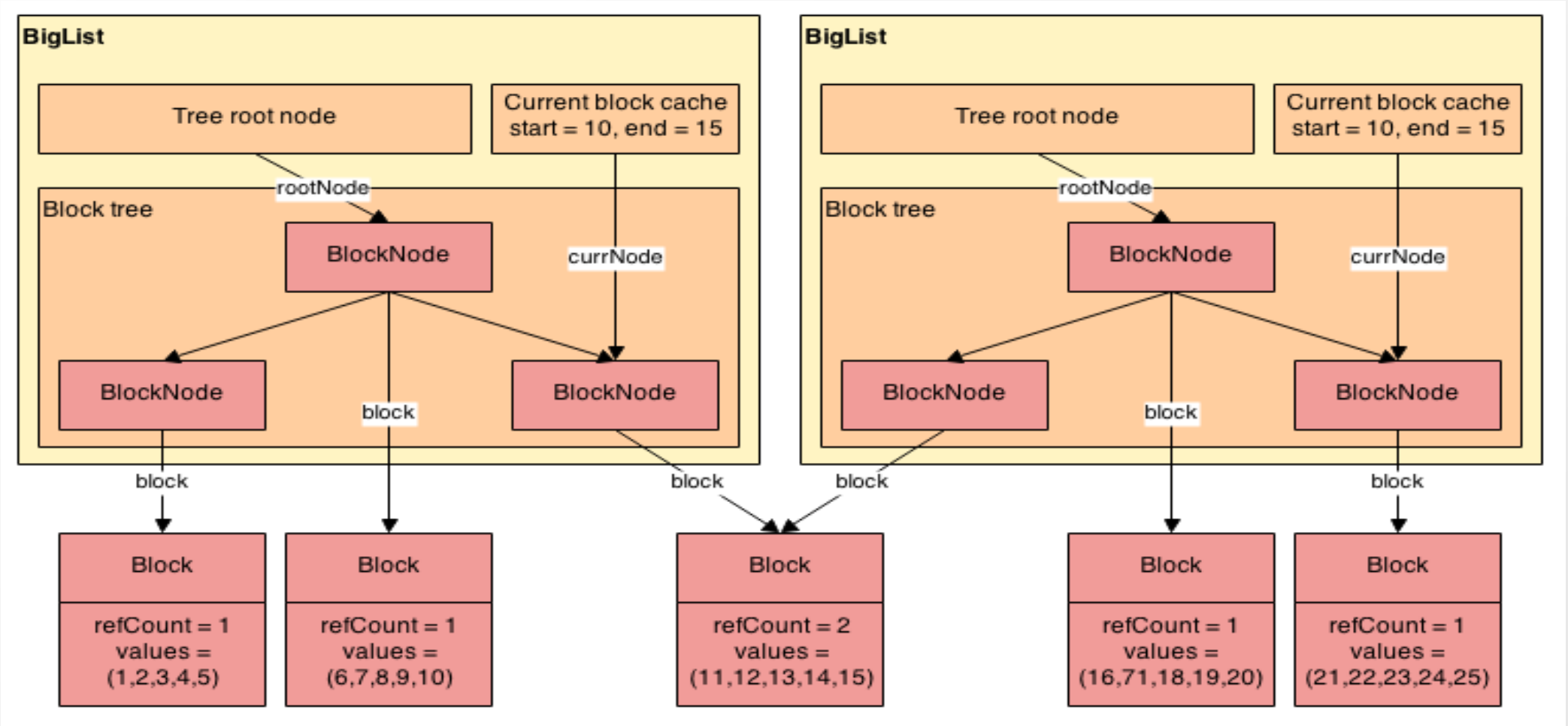
BigList: Block Tree

- For each operation, the affected block must be determined first
- Blocks are maintained in a specialized tree for fast access
- Information for the last accessed block is cached to exploit the locality of reference

BigList: Share Blocks

- Each block maintains a reference count to support a copy-on-write approach
- Initially blocks have reference count of 1 (block is private, modification allowed)
- If a list is copied, only the reference count needs to be incremented (block is shared, must be copied before modification)
- The reference count is automatically decremented again by the finalizer

BigList: Implementation



BigList: Operations

- Animations

BigList: Features

BigList also implements the IList interface

- setAll(index,coll) / setArray(index, elems...) / setMult(index,len,elem)
- Internal copying: copy / move, etc.
- External copying: transferCopy, etc.

BigList also has classes for primitive types

- IntBigList, IntObjBigList, etc.

Benchmarks

- List with 1'000'000 elements

	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

- And the winner is: BigList!

The Quest Has Come To An End!

Recipe:

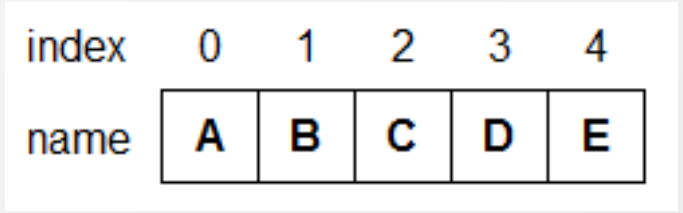
- Use GapList instead of ArrayList / LinkedList / ArrayDeque
- Use BigList instead of GapList if the list can become larger
- Measure, don't guess

And Now For Something Completely Different

Key Collections

Motivation

We need to maintain a list of columns with name:



- columns are ordered
- column names must be unique
- efficient access to the columns by both index and name

Which collection do you choose?
How long does it take to implement?

Solution

The column list:

- 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;
  }
  }
- UNSAFE!

Constraints

- class Table {
  List<Column> getCols() {
  return Collections.
  unmodifiableList(cols);
  }
  }
- READ-ONLY

Constraints

- ```
  class Table {
      void addColumn(Column col) {
          check(col);
          cols.add(col);
      }
  }
  ```
- POOR! or TEDIOUS!

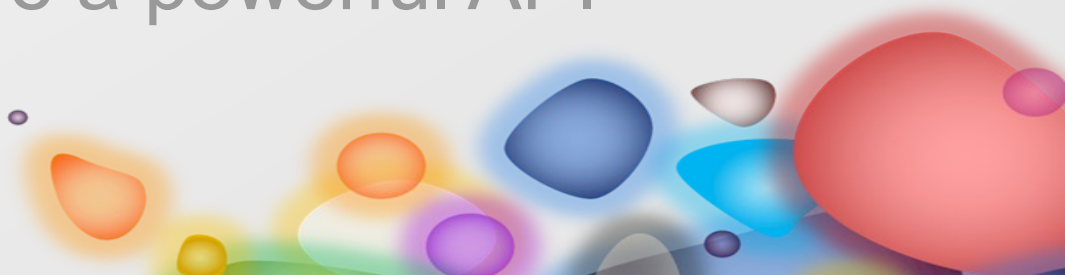

  ```

Constraints

- class Table {
  void setColumns(List<Column> cols) {
  check(cols);
  this.cols = cols;
  }
  }
- INEFFICIENT!

Constraints

- class Table {
  Key1List<Column,String> cols ...
  Key1List<Column,String> getCols() {
  return cols;
  }
  }
- EASY!
- Constraints are important to have a powerful API



Keys

- A key is a value extracted from an element using a mapper function
  - Key1List<Column,String> withKey1Map(Column::getName)
- With this approach, all key collections always handle a Collection of elements,
  no need to use a separate Map interface

Keys vs Maps

- Typical case:
  - Class Column { String key; Column col; }
- How the JDK handles the typical case:
  - Map<String,Column> cols;
    col = new Column("name");
    cols.put(col.getName(), col);
- How Key Collections handle the atypical case:
  - Class KeyColumn { String key; Column col; }

Key Map and Element Set

- The collection of all key values extracted by a function is called Key Map
  - withKey1Map
- Also the elements itself can be used as key, this is called the Element Set
  - withElemSet
- Keys can be used for fast access
  - getByKey1("A")

Keys and Constraints

Keys can be used for defining constraints:

- Null values: allowed / prohibited
  withKey1Null
- Duplicate values: allowed / prohibited
  withKey1Duplicates

Shortcuts:

- WithPrimaryKey1: not null, no duplicates
  withUniqueKey1: no duplicates except null

Keys and Sorting

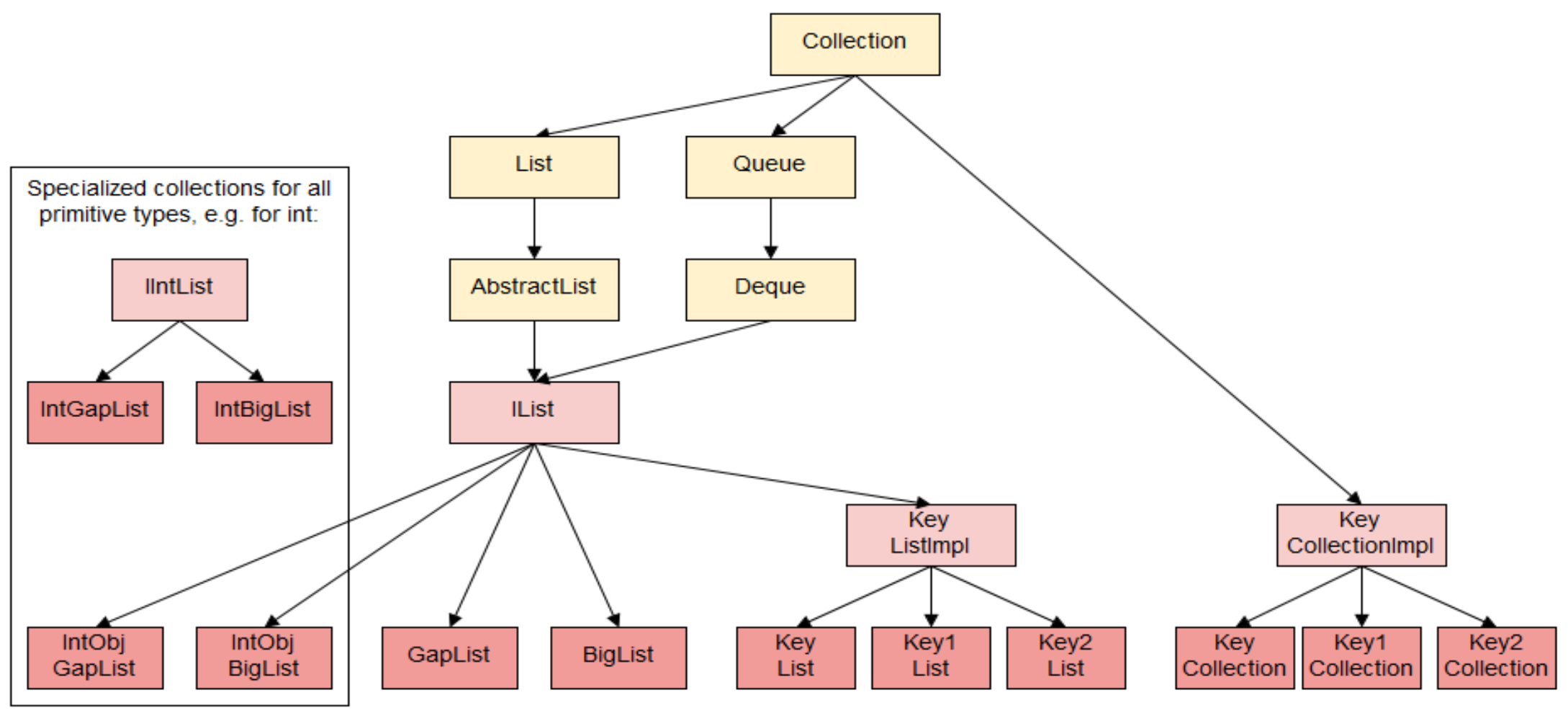
Keys can also be sorted:

- Sorting: the order of keys can be random or sorted (using natural order or a custom comparator)
- Sorting a key does not sort the collection itself
- Use withOrderBy for this

More Constraints

- Only positive numbers
  withConstraint(i -> i>=0)
- Define maximum size of collection
  withMaxSize(10)
- Define maximum size as window
  withWindowSize(10)
- Triggers
  withBeforeInsert / withAfterInsert
  withBeforeDelete / withAfterDelete

Classes


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 Overview

- KeyList classes implement the IList interface (i.e. List and Deque)
- KeyCollection classes implement the JDK Collection interface
- Use asSet() / asMap() for accessing the elements as JDK Set or Map

API Methods

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

API Usage

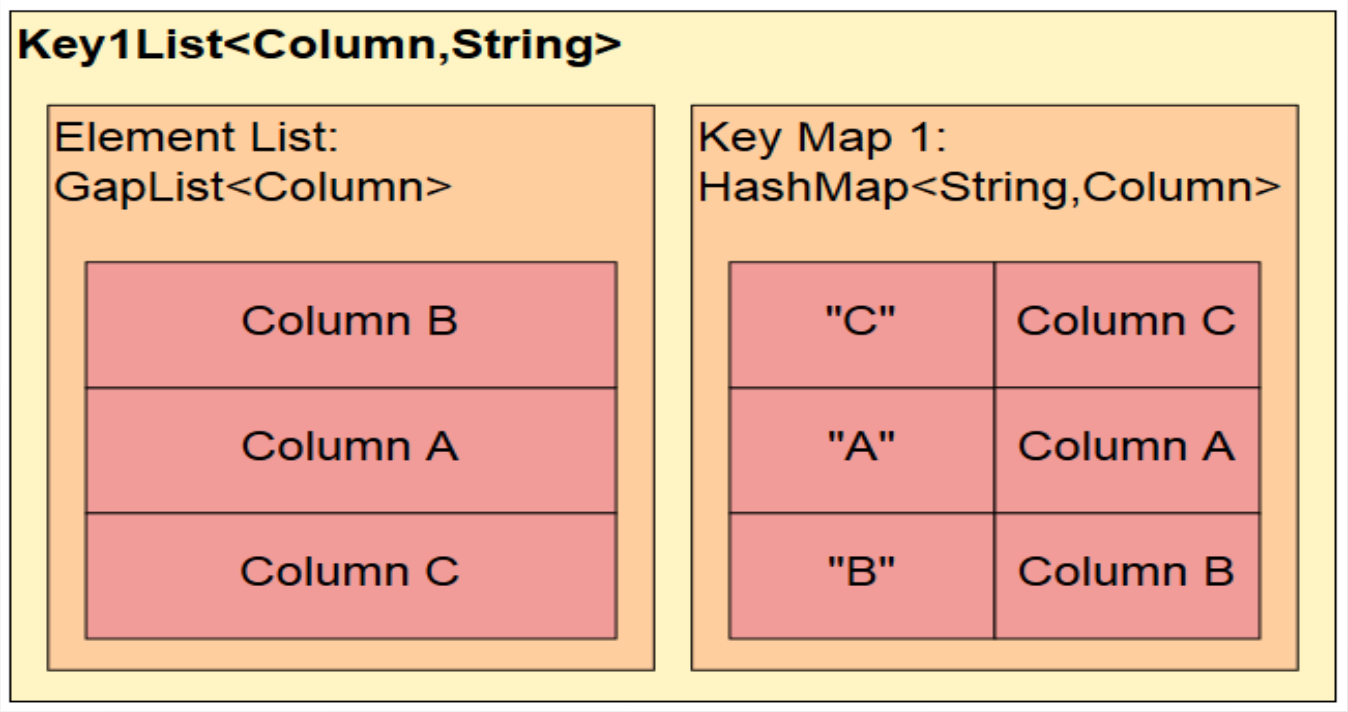
- cols.add(new Column("A"));
  - ok
- Column col = cols.getByKey1("A");
  - retrieve element
- cols.add(new Column("A"));
  - fails with DuplicateKeyException
- cols.putByKey1(new Column("A"))
  - replaces element
- cols.removeByKey("A")
  - removes element

Implementation

- Key Collections use other collections as building blocks:
  - GapList / BigList (or its primitive variants)
  - HashMap / TreeMap
- The needed components are chosen according to configuration if the collection is built

Layout

Column list example: needs GapList / HashMap



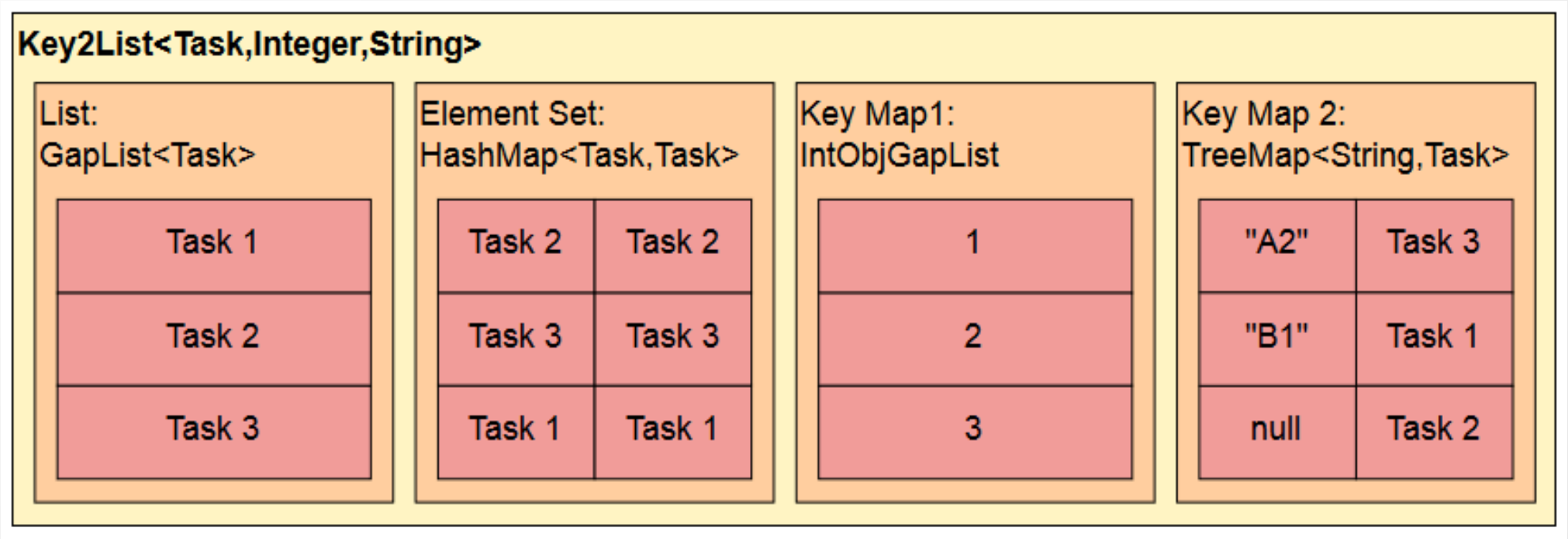
Removal by key needs iterating...

Example: 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

Needed Components:
GapList / HashMap / IntObjGapList / TreeMap



Example: Constraint List

- A List which accepts only positive integers

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

- A List which accepts only ten elements

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

Example: Set List

- SetList combines a list with fast access by element
- A Set List which accepts also null and duplicates

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

- A Set List which accepts only unique values

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

- Note that the element set is automatically created

Example: Sorted List

- A list of Files sorted by name
- Key1List<File,String> list =
  new Key1List.Builder<File,String>().
  withKey1Map(File::getName).
  withPrimaryKey1().withKey1OrderBy(true).build();
- But should there be something like a sorted list?

Example: Multi Set

- MultiSet stores only the count of occurrences for elements
- An unordered MultiSet
  new KeyCollection.Builder<String>.
  withElemCount().build()
- A sorted MultiSet
  new KeyCollection.Builder<String>.
  withElemCount().withElemSort(true).build()
- Option withElemCount() is only applicable for KeyCollection

Example: BiMap

- A BiMap where both key maps are sorted
- 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;
  }

Using Key Collections - Raw

- Type parameters are tedious to repeat
- Key1List<Column,String> cols =
  new Key1List.Builder<Column,String>.
  withKey1Map(Column::getName).
  withPrimaryKey1().build();
- void func(Key1List<Column,String> cols) {}

Using Key Collections - Refined

Create concrete type to hide type parameters

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

That's all folks!

www.magicwerk.org