C♭: A New Modular Approach to Implementing Efficient and Tunable Collections

Stephan Brandauer
Uppsala University

Elias Castegren
KTH

Tobias Wrigstad
Uppsala University

C♭: A New Modular Approach to Implementing Efficient and Tunable Collections

Stephan Brandauer
Uppsala University

Elias Castegren
KTH

Tobias Wrigstad
Uppsala University

Motivation

Motivation

- In the implementation of collections, the business logic and the data representation are intimately tangled

Motivation

- In the implementation of collections, the business logic and the data representation are intimately tangled
- Two implementations of the same interface are functionally equivalent but may differ on non-functional properties

  Prepend of a linked list is fast — prepend of an array list is not

  Random access in an array list is fast — random access in a linked list is not

  …

Motivation

- In the implementation of collections, the business logic and the data representation are intimately tangled
- Two implementations of the same interface are functionally equivalent but may differ on non-functional properties

  Prepend of a linked list is fast — prepend of an array list is not

  Random access in an array list is fast — random access in a linked list is not

  …

- In Java, for example, we abstract the concrete implementation (`LinkedList`, `ArrayList`) by programming against a `List` interface

  As we learn about the software we are developing, we can easily swap out concrete implementations

  Also possible ”just-in-time”, transitioning from prepend-phase to random access-phase, etc.

Implementation of LinkedList and ArrayList

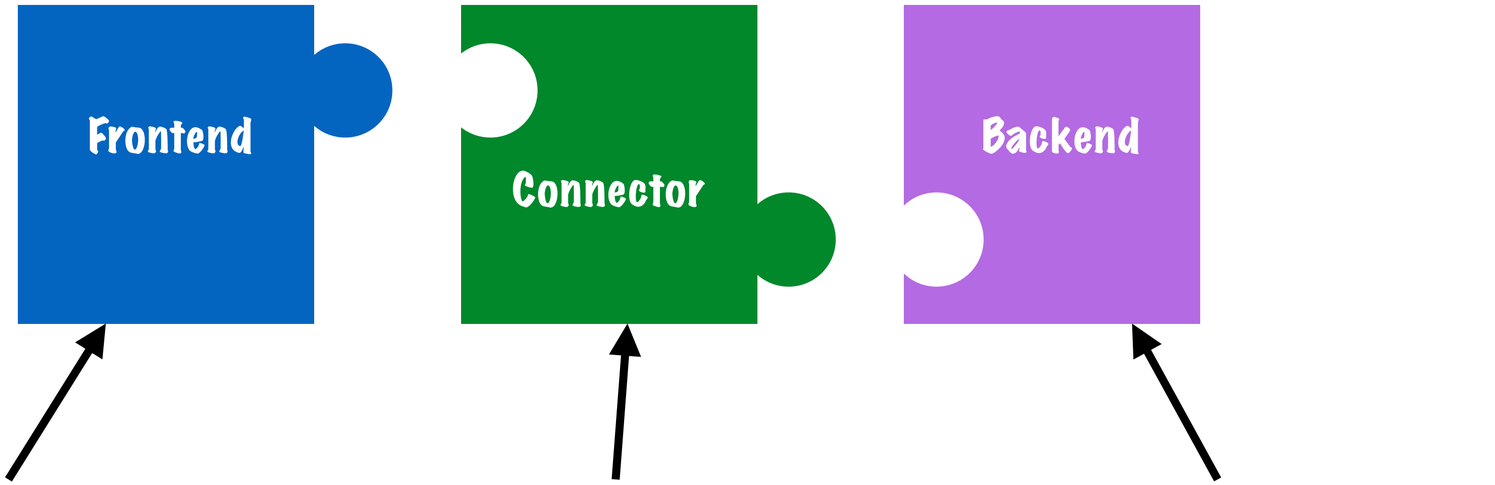
- They both implement the List interface with **41** methods
- `LinkedList` implemented by Josh Bloch, `ArrayList` implemented by Josh Bloch and Neal Gafter
- They share ~**5** methods (out of ~**50**) through inheritance from e.g., `AbstractList`

  Examples: `equals`, `hashCode`, `toString`, `containsAll`
- In the source code, there are more opportunities for reuse, but mostly ”ignored” for performance reasons because different data representation favor different access patterns

Numbers included to show magnitude

This Work is About the Separation of Data Representation from the Implementation of Functional Specifications

- Collections are implemented as three components working together:



Complete implementation of the **functional specification**

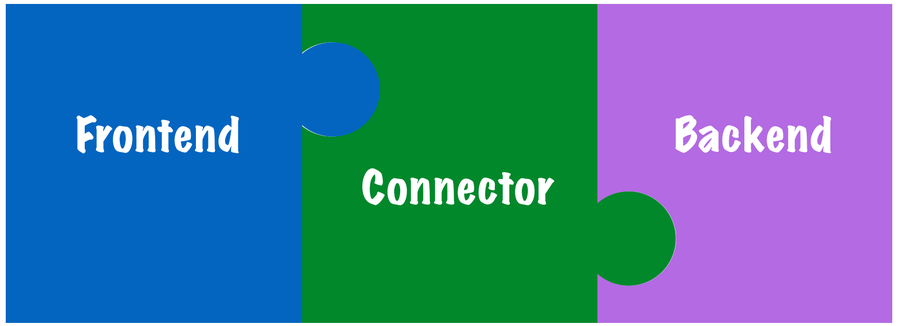
Abstract interface to the backend

Element **storage** — **tuned** for some scenarios (e.g., dense, sparse)

Restricted to collections whose spines can be described as trees

This Work is About the Separation of Data Representation from the Implementation of Functional Specifications

- Collections are implemented as three components working together:



- Frontends and backends can be combined freely

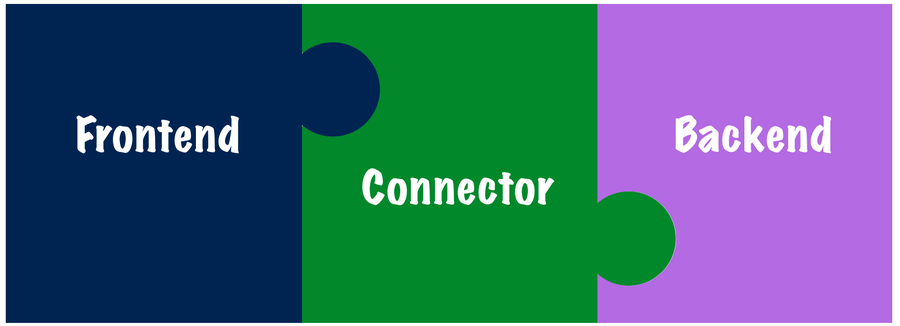
  `List` interface + Array-based backend = `ArrayList`

  Connector logic synthesized from an abstract specification

- Prototype Implementation in Java

This Work is About the Separation of Data Representation from the Implementation of Functional Specifications

- Collections are implemented as three components working together:



- Frontends and backends can be combined freely

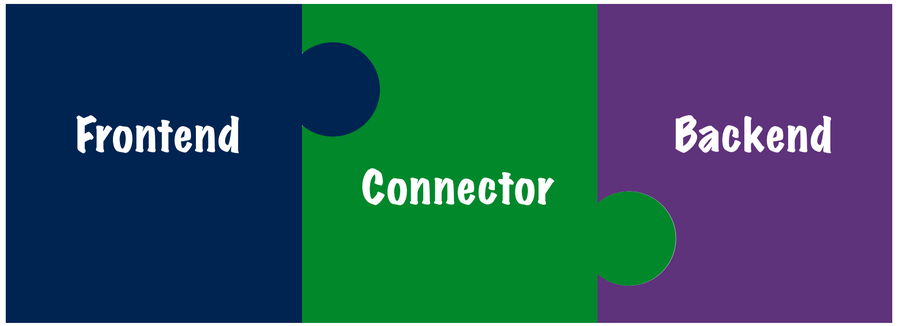
  `List` interface + Array-based backend = `ArrayList`

  Connector logic synthesized from an abstract specification

- Prototype Implementation in Java

This Work is About the Separation of Data Representation from the Implementation of Functional Specifications

- Collections are implemented as three components working together:



- Frontends and backends can be combined freely

  `List` interface + Array-based backend = `ArrayList`

  Connector logic synthesized from an abstract specification

- Prototype Implementation in Java

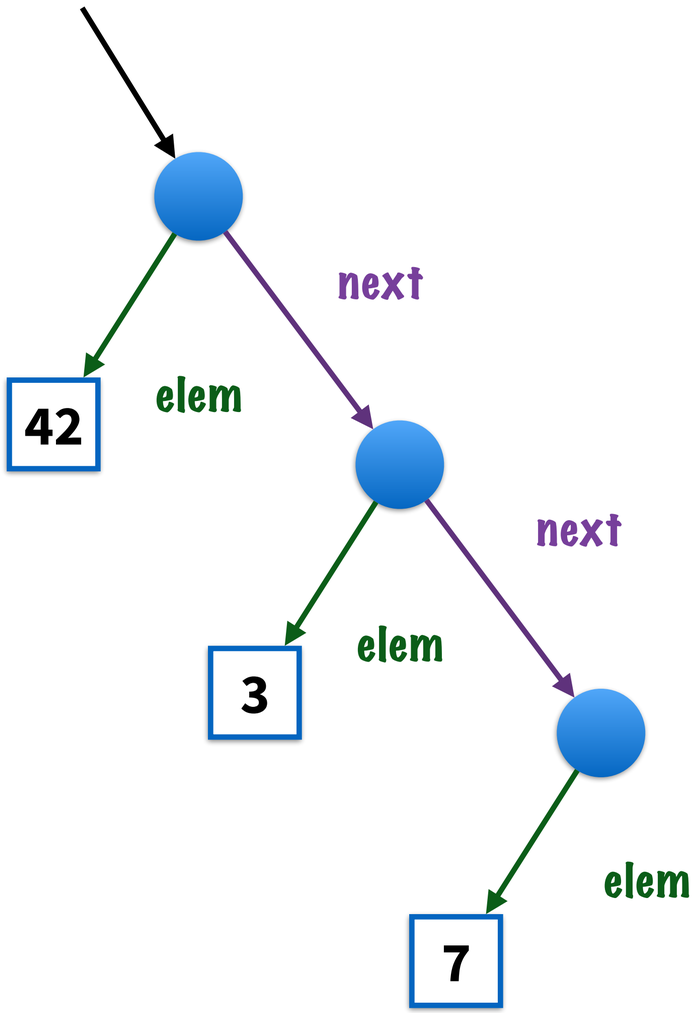
Connector Logic

- Front-ends access storage via a cursor, which can be thought of as a pointer variable with strict rules on how it may be reassigned

  The cursor is a robot that navigates the storage in the backend and retrieves or stores elements on their uniquely named storage locations

  How the cursor may move is specified as an automaton

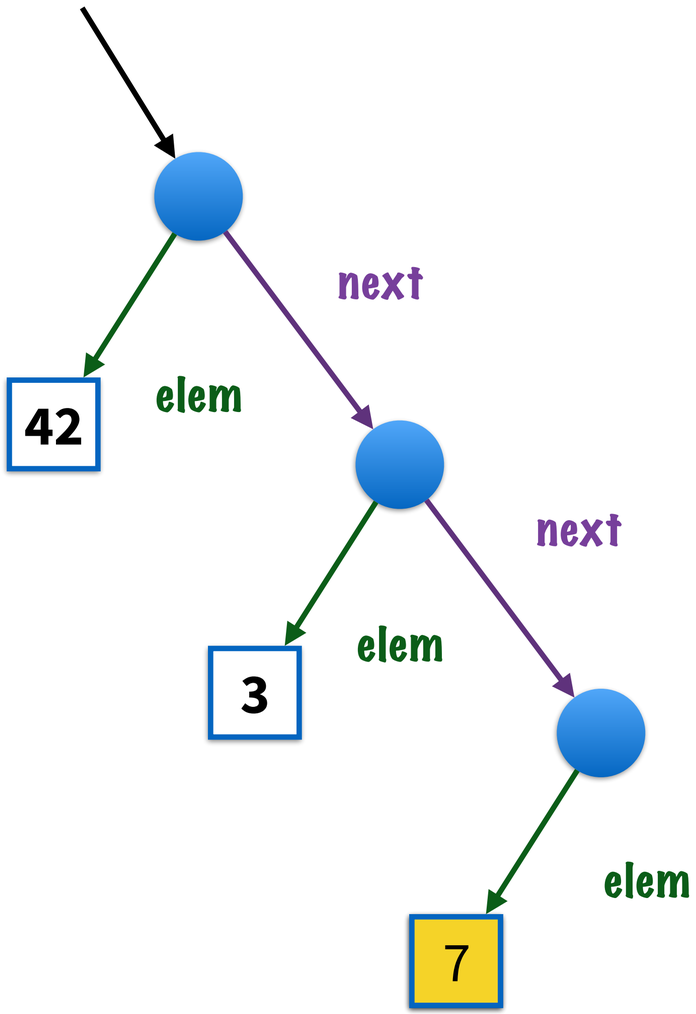
  This automaton is embedded in the source code as a Java annotation, cursor generated from it automatically



Logical representation of a sequence/list

Connector automaton:

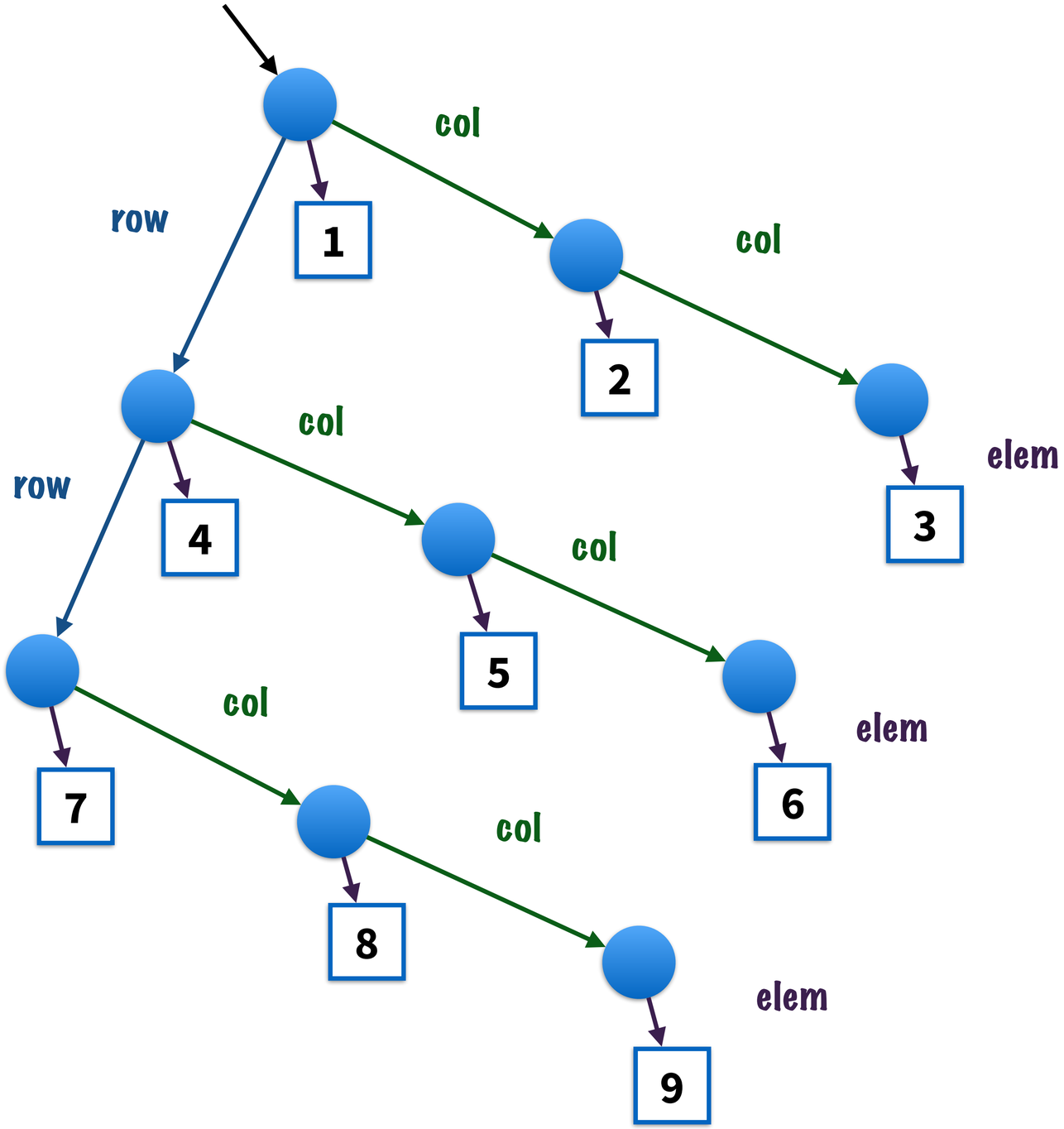
```
*(next) -> elem
```



Logical representation of a sequence/list

Connector automaton:

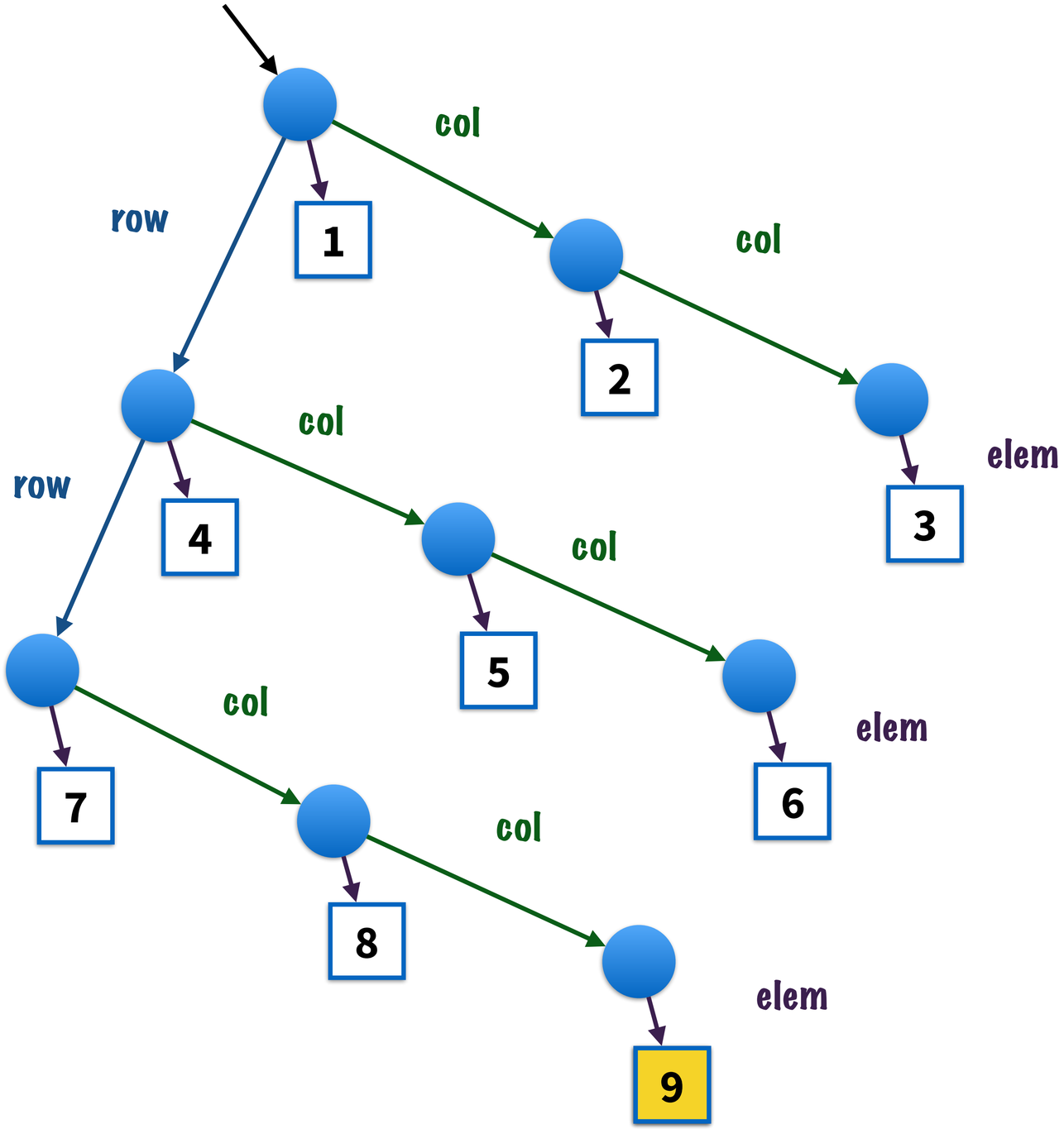
```
*(next) -> elem
```



Logical representation of a 3 x 3 matrix

Connector automaton:

```
*3(row) -> *3(col) -> elem
```



Logical representation of a 3 x 3 matrix

Connector automaton:

```
*3(row) -> *3(col) -> elem
```

~~Legal Robot Movement~~Cursor Automaton

C♭ expression	Interpretation	Example
`*(next) -> elem`	Sequence	*next.next.next.elem*
`*(‖(left,right)) -> elem`	2-ary tree	*left.right.left.elem*
`*(*7(child)-> pick) -> elem`	7-ary tree	*child.child.pick.child.pick.elem*
`*3(row) -> *3(col) -> elem`	3 x 3 matrix	*row.row.col.col.elem*

$C ::=$		*(C♭ expression.)*
	$\mid *(S) \to S'$	Repeat S unboundedly, then S'.
	$\mid *N(S) \to S'$	Repeat S upto N times, then S'.
	$\mid \ \|\|(A_1, \ldots, A_n)$	Pick one alternative in $\{A_1, \ldots, A_n\}$.

Backend Abstraction

- Each storage location has a unique path to it
- We can thus enumerate the paths and use these as ids

  For example, `row().row().col().col().elem()` on a previous slide will have id 8 — which nicely translates into an array index, or key

  The equivalent path `row(2).col(2).elem()` can be implemented in constant-time for backends that support it

- The key components of a backend are `get()` and `set()` operations that take legal paths as inputs

```
// Backend
Backend<T> backend = …

// Create a new cursor, positioned at root
Cursor c = new Cursor();

// Move the cursor to the right place
c.row().row().col().col();

// Extract the element
T value = backend.get(c.elem());
```

Backend Abstraction

- Each storage location has a unique path to it
- We can thus enumerate the paths and use these as ids

  For example, `row().row().col().col().elem()` on a previous slide will have id 8 — which nicely translates into an array index, or key

  The equivalent path `row(2).col(2).elem()` can be implemented in constant-time for backends that support it

- The key components of a backend are `get()` and `set()` operations that take legal paths as inputs

```
// Backend
Backend<T> backend = …

// Create a new cursor, positioned at root
Cursor c = new Cursor();

// Analogous to matrix[2][2] but abstract
c.row(2).col(2);

// Extract the element
T value = backend.get(c.elem());
```

Backend Abstraction

This call actually returns the int 8

- Each storage location has a unique path to it
- We can thus enumerate the paths and use these as ids

  For example, `row().row().col().col().elem()` on a previous slide will have id 8 — which nicely translates into an array index, or key

  The equivalent path `row(2).col(2).elem()` can be implemented in constant-time for backends that support it

- The key components of a backend are `get()` and `set()` operations that take legal paths as inputs

```
// Backend
Backend<T> backend = …

// Create a new cursor, positioned at root
Cursor c = new Cursor();

// Analogous to matrix[2][2] but abstract
c.row(2).col(2);

// Extract the element
T value = backend.get(c.elem());
```

Generation of Cursor Classes

- Generated at compile time by a library — no compiler modifications
- Encapsulates logic for calculating and returning the unique id for a path, represented as an integer
- Stateful — pointer variable analogy

  Given methods from the symbols in the C♭ expression — e.g., `left`, `right`, `next`, `elem` etc.

  Each call to a method will manipulate the id state

  `next()` = ++

  `left()` = current position * 2 + 1 *(binary heap layout)*

  `right()` = current position * 2 + 2

  `elem()` returns the path identifier

Implementing Collections using an Abstract Representation

```
@Cflat("*(next)->elem")
class Sequence<T> implements java.util.List {
    private Backend<T> data;
    private SequenceCursor tail = new SequenceCursor();

    boolean add(T obj) {

        data.set(this.tail.elem(), obj);

        tail.next();
        return true;
    }

    boolean contains(Object obj) {
        return data.usedLocations().findAny((T) obj) != -1;
    }
}
```

Implementing Collections using an Abstract Representation

```
@Cflat("*(next)->elem")
class Sequence<T> implements java.util.List {
    private Backend<T> data;
    private SequenceCursor tail = new SequenceCursor();

    boolean add(T obj) {

        data.set(this.tail.elem(), obj);

        tail.next();
        return true;
    }

    boolean contains(Object obj) {
        return data.usedLocations().findAny((T) obj) != -1;
    }
}
```

Implementing Collections using an Abstract Representation

```
@Cflat("*(next)->elem")
class Sequence<T> implements java.util.List {
    private Backend<T> data;
    private SequenceCursor tail = new SequenceCursor();

    boolean add(T obj) {

        data.set(this.tail.elem(), obj);

        tail.next();
        return true;
    }

    boolean contains(Object obj) {
        return data.usedLocations().findAny((T) obj) != -1;
    }
}
```

Implementing Collections using an Abstract Representation

```
@Cflat("*(next)->elem")
class Sequence<T> implements java.util.List {
    private Backend<T> data;
    private SequenceCursor tail = new SequenceCursor();

    boolean add(T obj) {

        data.set(this.tail.elem(), obj);

        tail.next();
        return true;
    }

    boolean contains(Object obj) {
        return data.usedLocations().findAny((T) obj) != -1;
    }
}
```

Implementing Collections using an Abstract Representation

```
@Cflat("*(next)->elem")
class Sequence<T> implements java.util.List {
    private Backend<T> data;
    private SequenceCursor tail = new SequenceCursor();

    boolean add(T obj) {

        data.set(this.tail.elem(), obj);

        tail.next();
        return true;
    }

    boolean contains(Object obj) {
        return data.usedLocations().findAny((T) obj) != -1;
    }
}
```

Implementing Collections using an Abstract Representation

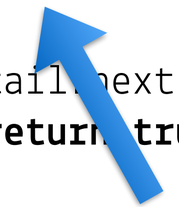
```
@Cflat("*(next)->elem")
class Sequence<T> implements java.util.List {
    private Backend<T> data;
    private SequenceCursor tail = new SequenceCursor();

    boolean add(T obj) {

        data.set(this.tail.elem(), obj);

        tail.next();
        return true;
    }

    boolean contains(Object obj) {
        return data.usedLocations().findAny((T) obj) != -1;
    }
}
```



Implementing Collections using an Abstract Representation

```
@Cflat("*(next)->elem")
class Sequence<T> implements java.util.List {
    private Backend<T> data;
    private SequenceCursor tail = new SequenceCursor();

    boolean add(T obj) {

        data.set(this.tail.elem(), obj);

        tail.next();
        return true;
    }

    boolean contains(Object obj) {
        return data.usedLocations().findAny((T) obj) != -1;
    }
}
```

Implementing Collections using an Abstract Representation

```
@Cflat("*(next)->elem")
class Sequence<T> implements java.util.List {
    private Backend<T> data;
    private SequenceCursor tail = new SequenceCursor();

    boolean add(T obj) {

        data.set(this.tail.elem(), obj);

        tail.next();
        return true;
    }

    boolean contains(Object obj) {
        return data.usedLocations().findAny((T) obj) != -1;
    }
}
```

Backend Implementation

- Manually implemented, following a `Backend` interface with 2 abstract methods and 28 default implementations

get an element for a path

set an element for a path

```
public interface Backend<T> {
    public abstract T get(int i);
    public abstract Backend<T> set(int i, T x);
}
```

28 methods for free based on these, e.g.:

```
has, forEach, forEachNonNull,
nonNullIndices, findFirst, findLast,
findAny, clearAll, …
```

Backend Implementations

- Manually implemented, following a `Backend` interface with 2 abstract methods and 28 default implementations

Hash table storage — using path ids as keys	Sparse
Immutable storage: using path ids as keys in an immutable map	Cheap copies
Indexed storage: like hash table storage, but with a backwards-map to permit fast location lookups of *values*	Fast way to find position by hash code
Sorted array storage: one array of non-null indices pointing into second array of values	Sparse array with efficient in-order iteration
Trie storage — uses a bitwise trie, four levels deep, using path ids as keys	Fast access and insert, sparse
Array storage — uses oath ids as array indices	Fast access and insert, dense
Reverse array storage — uses path ids as array indices	Like Array but reverse the data order

Backend Implementations

- Manually implemented, following a `Backend` interface with 2 abstract methods and 28 default implementations

Hash table storage — using path ids as keys	Sparse
Immutable storage: using path ids as keys in an immutable map	Cheap copies
Indexed storage: like hash table storage, but with a backwards-map to permit fast location lookups of *values*	Fast way to find position by hash code
Sorted array storage: one array of non-null indices pointing into second array of values	Sparse array with efficient in-order iteration
Trie storage — uses a bitwise trie, four levels deep, using path ids as keys	Fast access and insert, sparse
Array storage — uses path ids as array indices	Fast access and insert, dense
Reverse array storage — uses path ids as array indices	Like Array but reverse the data order

Backend Implementations

- Manually implemented, following a `Backend` interface with 2 abstract methods and 28 default implementations

Hash table storage — using path ids as keys	Sparse
Immutable storage: using path ids as keys in an immutable map	Cheap copies
Indexed storage: like hash table storage, but with a backwards-map to permit fast location lookups of *values*	Fast way to find position by hash code
Sorted array storage: one array of non-null indices pointing into second array of values	Sparse array with efficient in-order iteration
Trie storage — uses a bitwise trie, four levels deep, using path ids as keys	Fast access and insert, sparse
Array storage — uses path ids as array indices	Fast access and insert, dense
Reverse array storage — uses path ids as array indices	Like Array but reverse the data order

Backend Implementations

- Manually implemented, following a `Backend` interface with 2 abstract methods and 28 default implementations

Hash table storage — using path ids as keys	Sparse
Immutable storage: using path ids as keys in an immutable map	Cheap copies
Indexed storage: like hash table storage, but with a backwards-map to permit fast location lookups of *values*	Fast way to find position by hash code
Sorted array storage: one array of non-null indices pointing into second array of values	Sparse array with efficient in-order iteration
Trie storage — uses a bitwise trie, four levels deep, using path ids as keys	Fast access and insert, sparse
Array storage — uses path ids as array indices	Fast access and insert, dense
Reverse array storage — uses path ids as array indices	Like Array but reverse the data order

Backend Implementations

- Manually implemented, following a `Backend` interface with 2 abstract methods and 28 default implementations

Hash table storage — using path ids as keys	Sparse
Immutable storage: using path ids as keys in an immutable map	Cheap copies
Indexed storage: like hash table storage, but with a backwards-map to permit fast location lookups of *values*	Fast way to find position by hash code
Sorted array storage: one array of non-null indices pointing into second array of values	Sparse array with efficient in-order iteration
Trie storage — uses a bitwise trie, four levels deep, using path ids as keys	Fast access and insert, sparse
Array storage — uses path ids as array indices	Fast access and insert, dense
Reverse array storage — uses path ids as array indices	Like Array but reverse the data order

Backend Implementations

- Manually implemented, following a `Backend` interface with 2 abstract methods and 28 default implementations

Hash table storage — using path ids as keys	Sparse
Immutable storage: using path ids as keys in an immutable map	Cheap copies
Indexed storage: like hash table storage, but with a backwards-map to permit fast location lookups of *values*	Fast way to find position by hash code
Sorted array storage: one array of non-null indices pointing into second array of values	Sparse array with efficient in-order iteration
Trie storage — uses a bitwise trie, four levels deep, using path ids as keys	Fast access and insert, sparse
Array storage — uses path ids as array indices	Fast access and insert, dense
Reverse array storage — uses path ids as array indices	Like Array but reverse the data order

Backend Implementations

- Manually implemented, following a `Backend` interface with 2 abstract methods and 28 default implementations

Hash table storage — using path ids as keys	Sparse
Immutable storage: using path ids as keys in an immutable map	Cheap copies
Indexed storage: like hash table storage, but with a backwards-map to permit fast location lookups of *values*	Fast way to find position by hash code
Sorted array storage: one array of non-null indices pointing into second array of values	Sparse array with efficient in-order iteration
Trie storage — uses a bitwise trie, four levels deep, using path ids as keys	Fast access and insert, sparse
Array storage — uses path ids as array indices	Fast access and insert, dense
Reverse array storage — uses path ids as array indices	Like Array but reverse the data order

Backend Implementations

- Manually implemented, following a `Backend` interface with 2 abstract methods and 28 default implementations

Hash table storage — using path ids as keys	Sparse
Immutable storage: using path ids as keys in an immutable map	Cheap copies
Indexed storage: like hash table storage, but with a backwards-map to permit fast location lookups of *values*	Fast way to find position by hash code
Sorted array storage: one array of non-null indices pointing into second array of values	Sparse array with efficient in-order iteration
Trie storage — uses a bitwise trie, four levels deep, using path ids as keys	Fast access and insert, sparse
Array storage — uses path ids as array indices	Fast access and insert, dense
Reverse array storage — uses path ids as array indices	Like Array but reverse the data order

Frontend Implementations

- Three implementations for our case study: `Sequence`, `Map` and `Matrix`

  Sequence: implements all methods in Java's `List` interface with ~4 SLOC per method

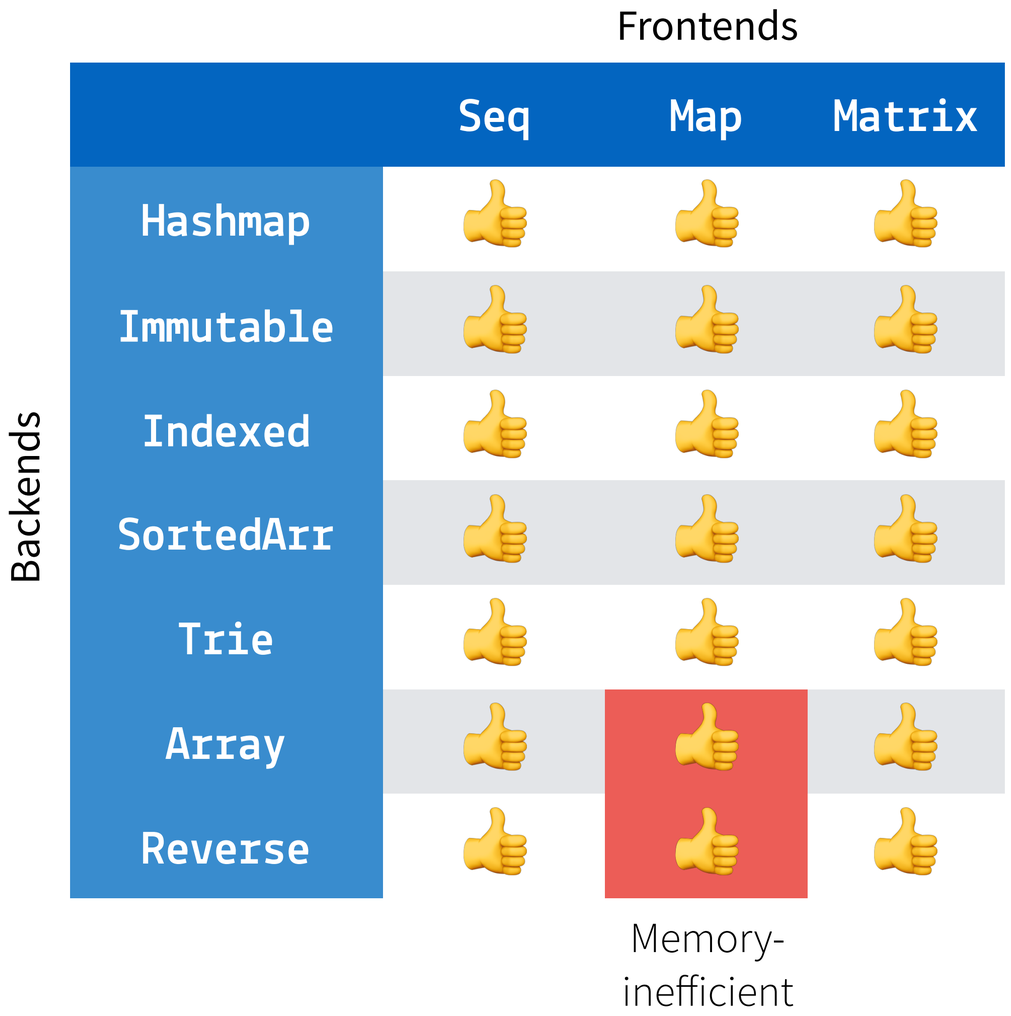
  Map: implementation of `AbstractMap` (25 methods) in 87 SLOC

  Matrix: *see paper for details — no standard Java matrix exist*

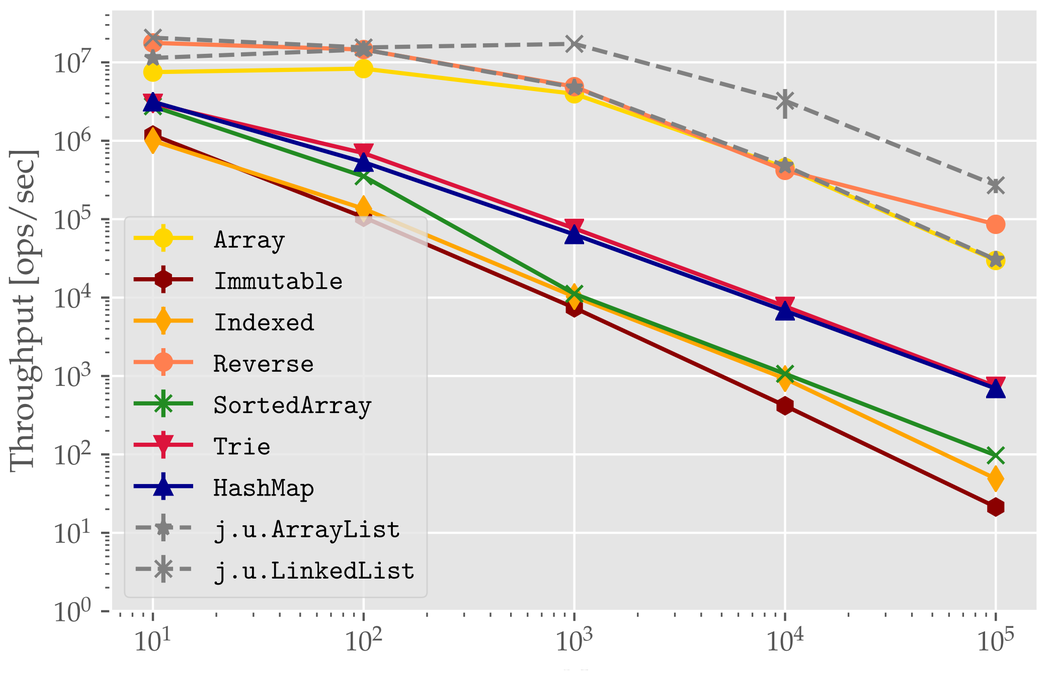
☠ **Anecdotal evidence!** ☠

C♭ can be used to implement full-featured collections

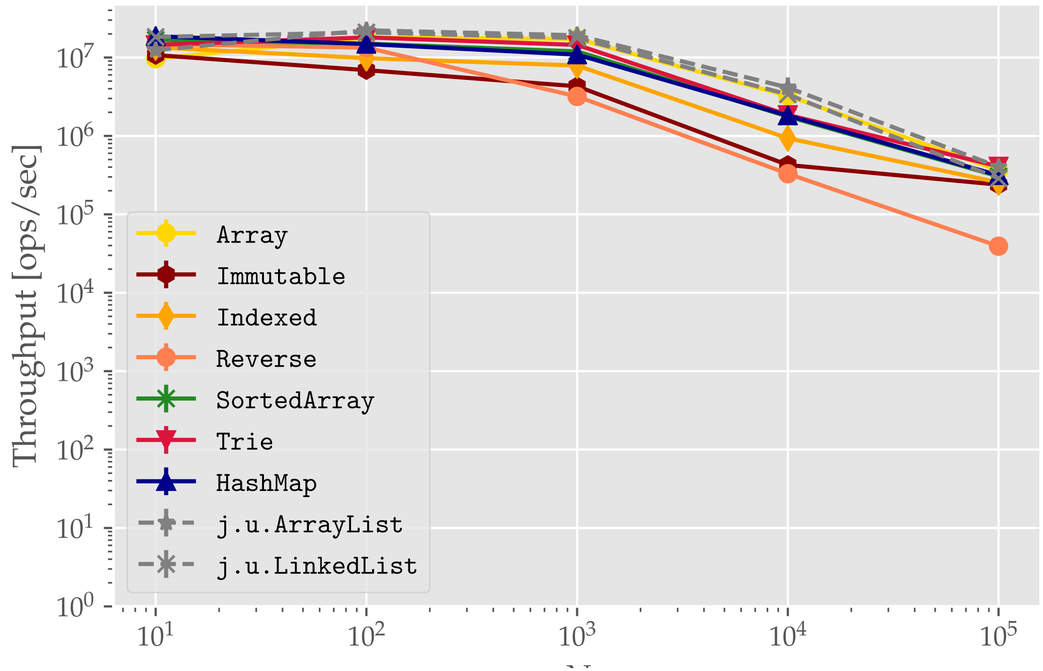
C♭ implementations are reasonably straightforward



Performance Compared to Java Collections

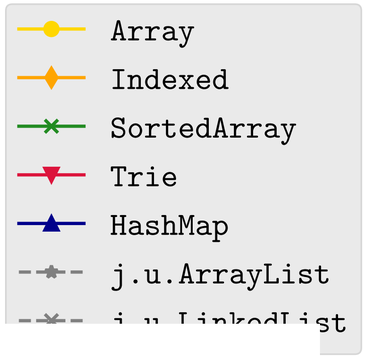


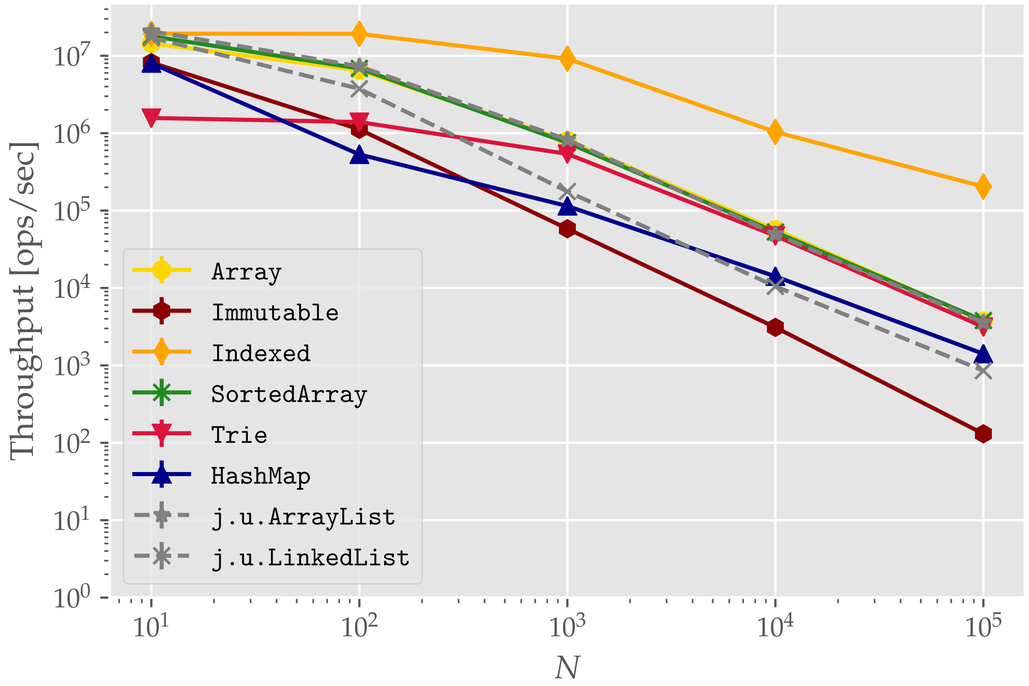
Insertion at front of a list



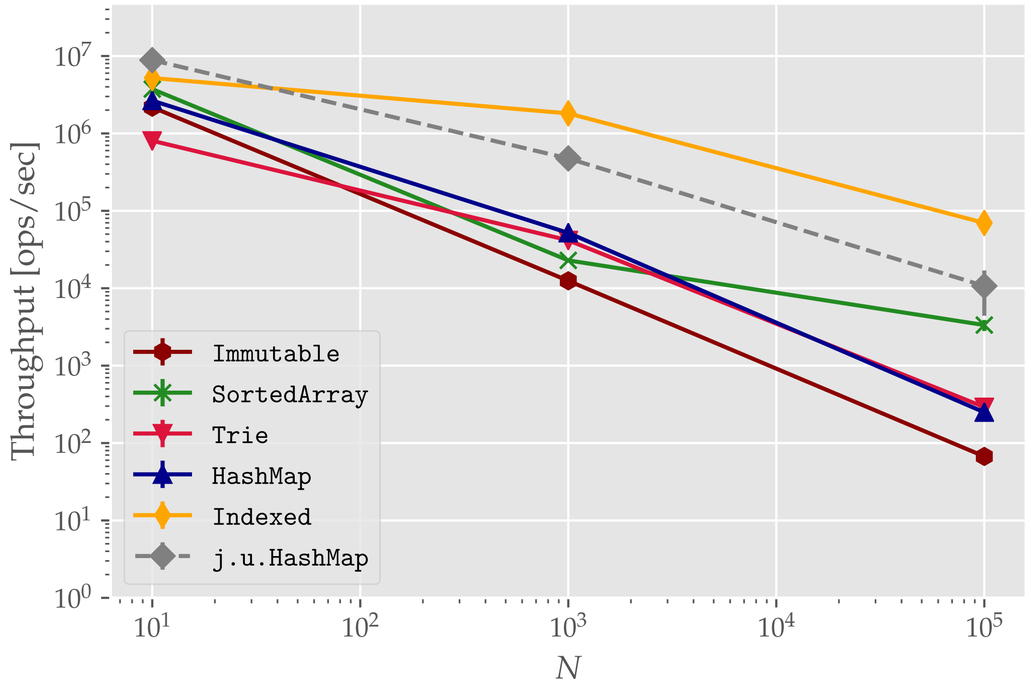
Insertion at end of a list

Performance Compared to Java Collections



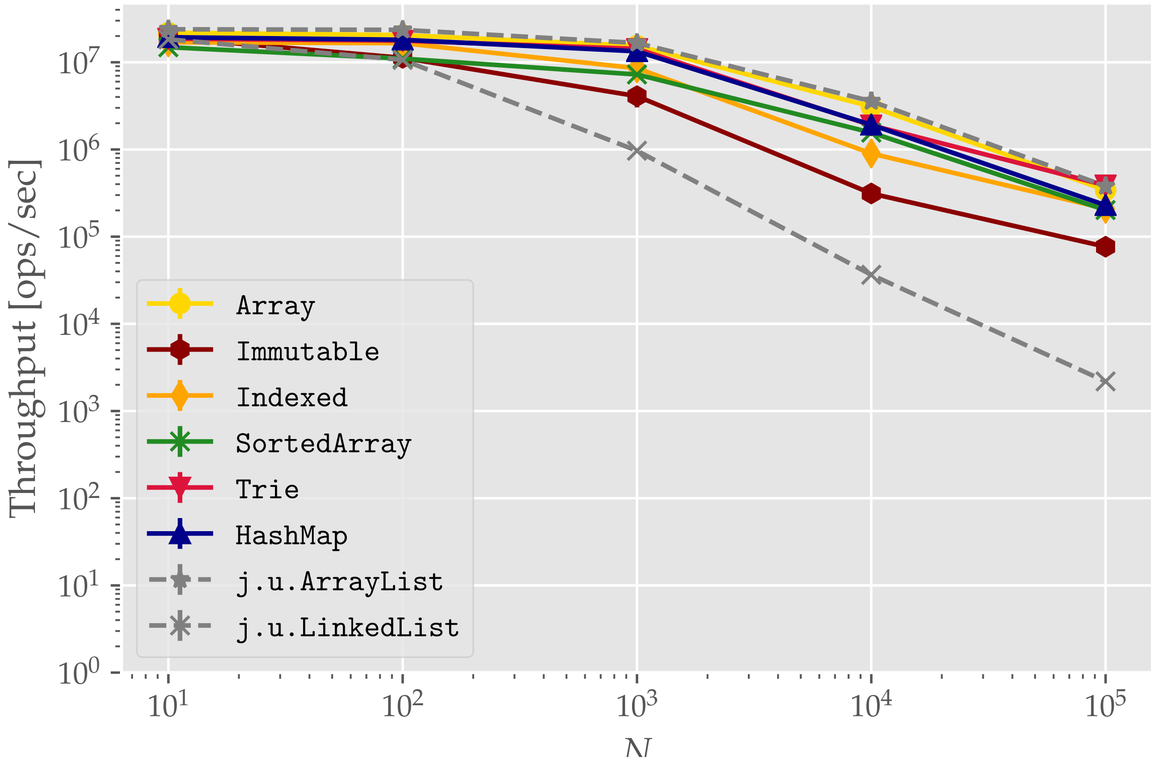


Searching for non-existing value in a list



Searching for a non-existing value in a map

Performance Compared to Java Collections



Random Access

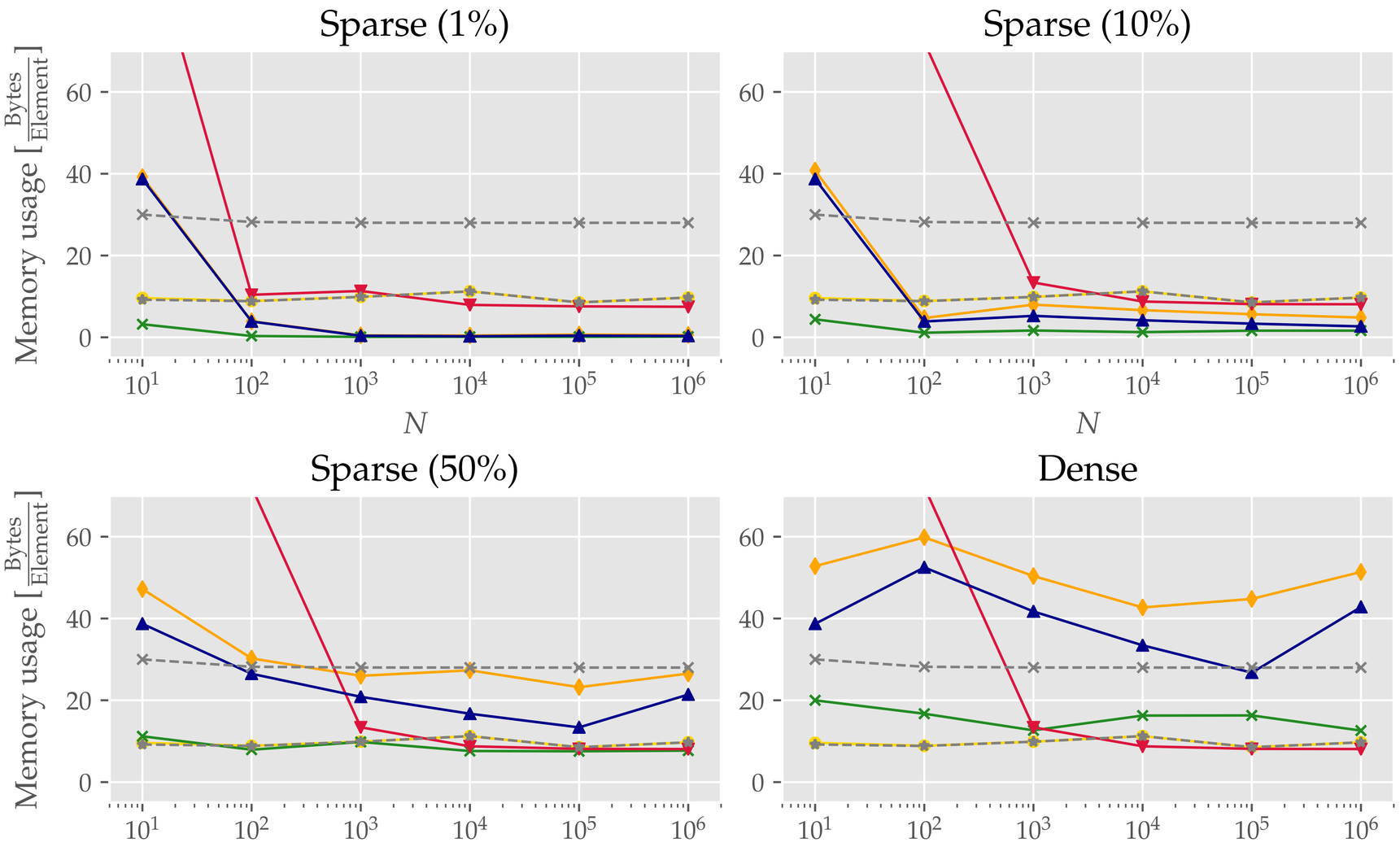
Array
Indexed
SortedArray
Trie
HashMap
j.u.ArrayList
j.u.LinkedList

☠ **Conclusion:** ☠

Even though we insert an extra level of indirection for operations, performance is still roughly on par with native Java implementations.

(Naturally not for all backends!)

Also in the Paper: Tunability



Sequence sparseness vs. length vs. memory usage.

In Summary

- We have proposed a new way for implementing collection data structures that separates functional specification from data representation

In Summary

- We have proposed a new way for implementing collection data structures that separates functional specification from data representation
- Frontends and backends can be combined freely

  Allows backend optimizations to be reused across many collections

  Allows tuning collections and reusing new backends with existing frontends

In Summary

- We have proposed a new way for implementing collection data structures that separates functional specification from data representation
- Frontends and backends can be combined freely

  Allows backend optimizations to be reused across many collections

  Allows tuning collections and reusing new backends with existing frontends

- A simple automaton that controls how the frontend may navigate the space of the backend allows us to automatically generate the code for connecting any frontend with any backend

In Summary

- We have proposed a new way for implementing collection data structures that separates functional specification from data representation
- Frontends and backends can be combined freely

  Allows backend optimizations to be reused across many collections

  Allows tuning collections and reusing new backends with existing frontends

- A simple automaton that controls how the frontend may navigate the space of the backend allows us to automatically generate the code for connecting any frontend with any backend
- We believe the code is straightforward, although it is initially unnatural if you are used to implementing data structures in the usual style

In Summary

- We have proposed a new way for implementing collection data structures that separates functional specification from data representation
- Frontends and backends can be combined freely

  Allows backend optimizations to be reused across many collections

  Allows tuning collections and reusing new backends with existing frontends

- A simple automaton that controls how the frontend may navigate the space of the backend allows us to automatically generate the code for connecting any frontend with any backend
- We believe the code is straightforward, although it is initially unnatural if you are used to implementing data structures in the usual style
- Our simple design scales to implementing the full interfaces of standard Java collections (only 2 methods needed in a backend)

In Summary

- We have proposed a new way for implementing collection data structures that separates functional specification from data representation
- Frontends and backends can be combined freely

  Allows backend optimizations to be reused across many collections

  Allows tuning collections and reusing new backends with existing frontends

- A simple automaton that controls how the frontend may navigate the space of the backend allows us to automatically generate the code for connecting any frontend with any backend
- We believe the code is straightforward, although it is initially unnatural if you are used to implementing data structures in the usual style
- Our simple design scales to implementing the full interfaces of standard Java collections (only 2 methods needed in a backend)

Thank you!