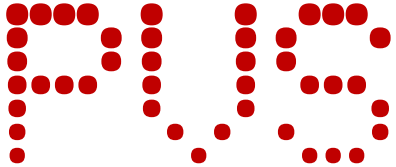
Empirical Study of Usage and Performance of Java Collections

[1]Diego Costa, [1]Artur Andrzejak, [1]Janos Seboek, [2]David Lo

[1]Heidelberg University, [2]Singapore Management University



Empirical Study of Usage and Performance of Java Collections

[1]Diego Costa, [1]Artur Andrzejak, [1]Janos Seboek, [2]David Lo

[1]Heidelberg University, [2]Singapore Management University

Published as:

Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. 2017. Empirical Study of Usage and Performance of Java Collections. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17), L'Aquila, Italy — April 22 - 26, 2017

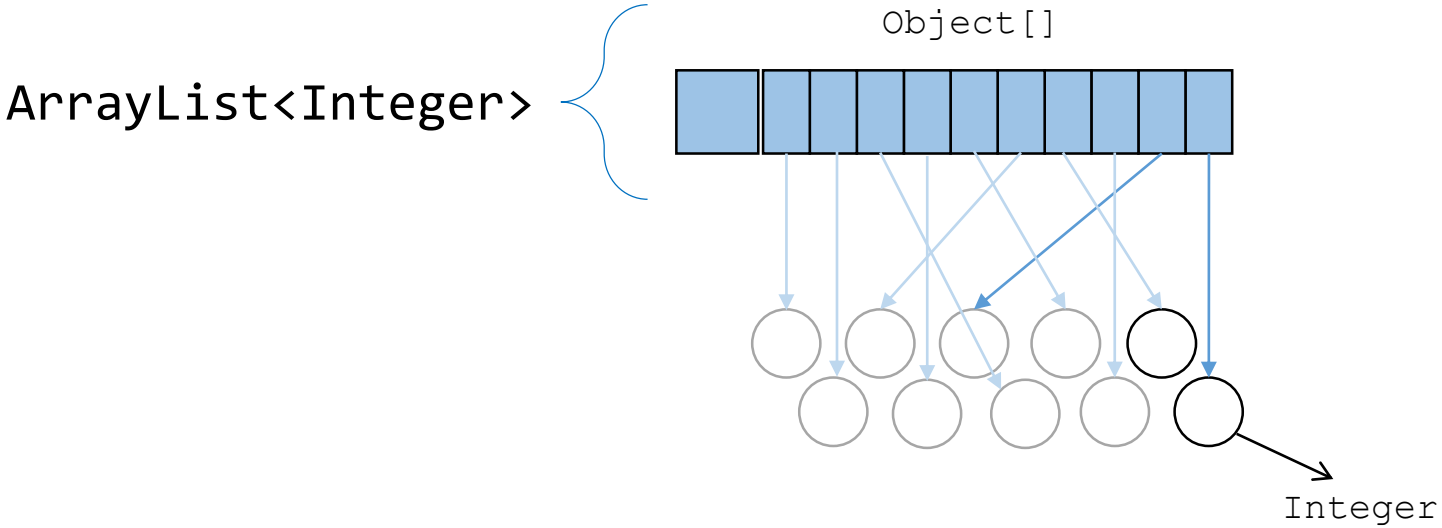
Paper/slides available at: https://pvs.ifi.uni-heidelberg.de/publications/

Collections

- Collections are objects that **groups** multiple elements into a single unit.
  - Use its metadata to track, access and manipulate its elements

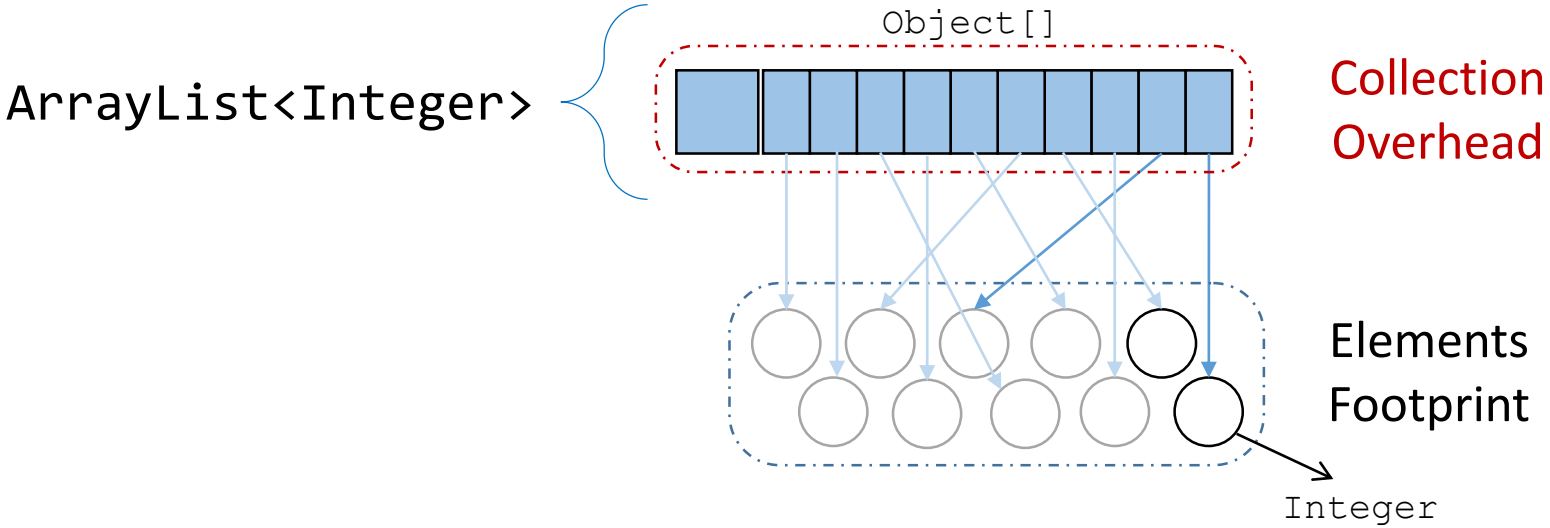
Collections

- Collections are objects that **groups** multiple elements into a single unit.
  - Use its metadata to track, access and manipulate its elements



Collections

- Collections are objects that **groups** multiple elements into a single unit.
  - Use its metadata to track, access and manipulate its elements



Motivation

- Numerous studies have identified the inefficient use of collections as the main cause of runtime bloat

Execution Time	Memory Usage	Energy Consumption
+17% Improv.	**+54% Improv.**	**+300% Improv.**
Configuration of one HashMap instance	Use of ArrayMaps instead of HashMaps	Use of ArrayList instead of LinkedList
[Liu et al. 2009]	[Ohad et al. 2009]	[Jung et al. 2016]

Collection Frameworks

- The Java Collection Framework offers a **Standard** implementation of the major collection abstractions
  - Stable and reliable framework
  - Easy to use
- However, there exist alternative libraries that provide a myriad of different implementations:
  - Primitive Collections (`IntArrayList`)
  - Immutable Collections
  - Multimaps (`Map<K, Collection>`)
  - Multisets (`Map<K, int>`)

Collection Frameworks

- The Java Collection Framework offers a **Standard** implementation of the major collection abstractions
  - Stable and reliable framework
  - Easy to use
- However, there exist alternative libraries that provide a myriad of different implementations:
  - Primitive Collections (`IntArrayList`)
  - Immutable Collections
  - Multimaps (`Map<K, Collection>`)
  - Multisets (`Map<K, int>`)

} Unsupported features

Collection Frameworks

- The Java Collection Framework offers a **Standard** implementation of the major collection abstractions
  - Stable and reliable framework
  - Easy to use
- However, there exist alternative libraries that provide a myriad of different implementations:
  - Primitive Collections (`IntArrayList`)
  - Immutable Collections
  - Multimaps (`Map<K, Collection>`)
  - Multisets (`Map<K, int>`)

Unsupported features

Simplified API

Analysis of Performance Impact of Alternative Collections

Goal: Can we find alternatives to the Standard collection types which improve performance on time/memory?

1. Study on Collections Usage
   - How often do programmers use alternative implementations?

2. Experimental Evaluation of popular Java Collection Libraries
   - Are there better alternatives to the most commonly used Collections regarding performance?

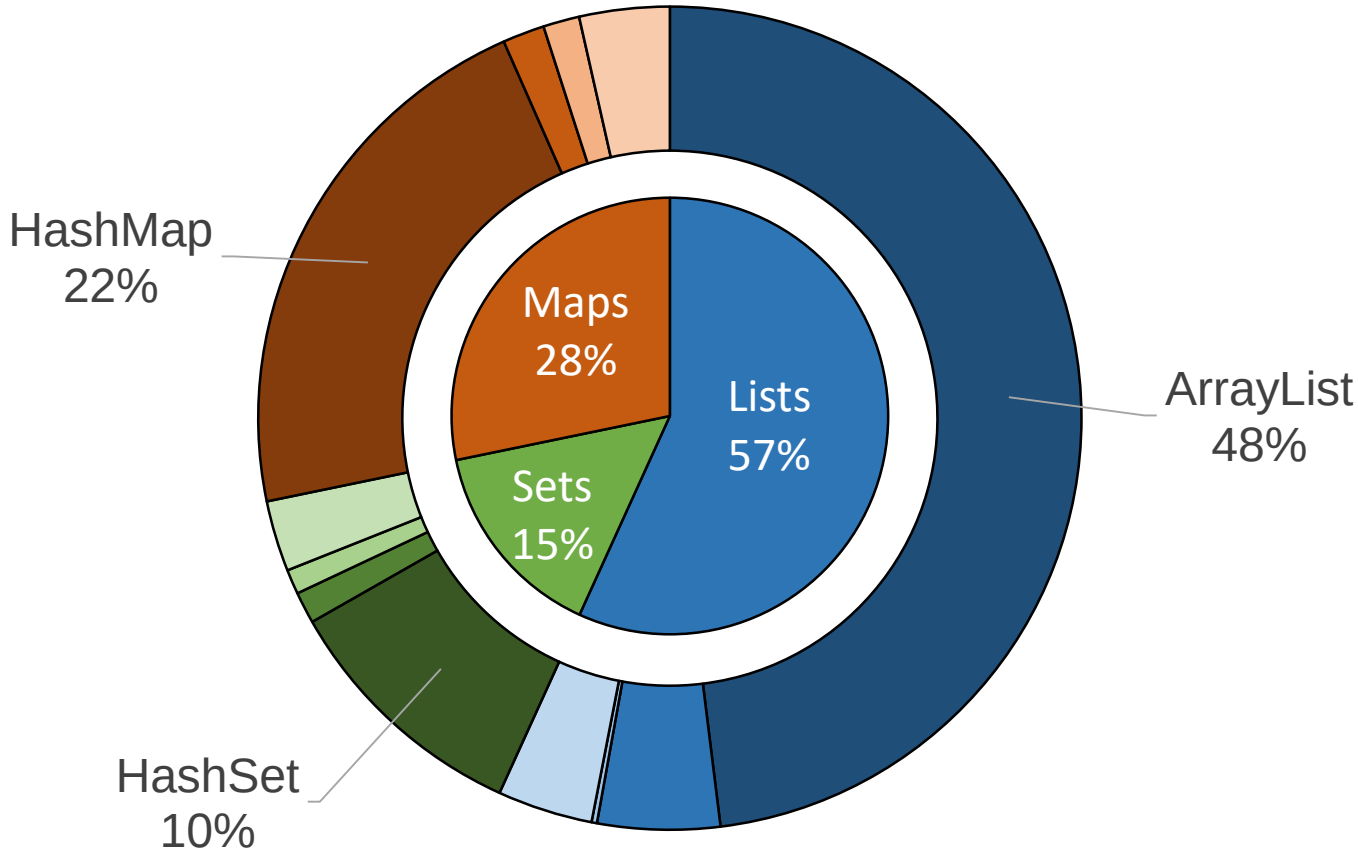
Study on Usage of Collections

- Dataset
  - We analyze the **GitHub Java Corpus**
    - 10K projects
    - 268 MLOC

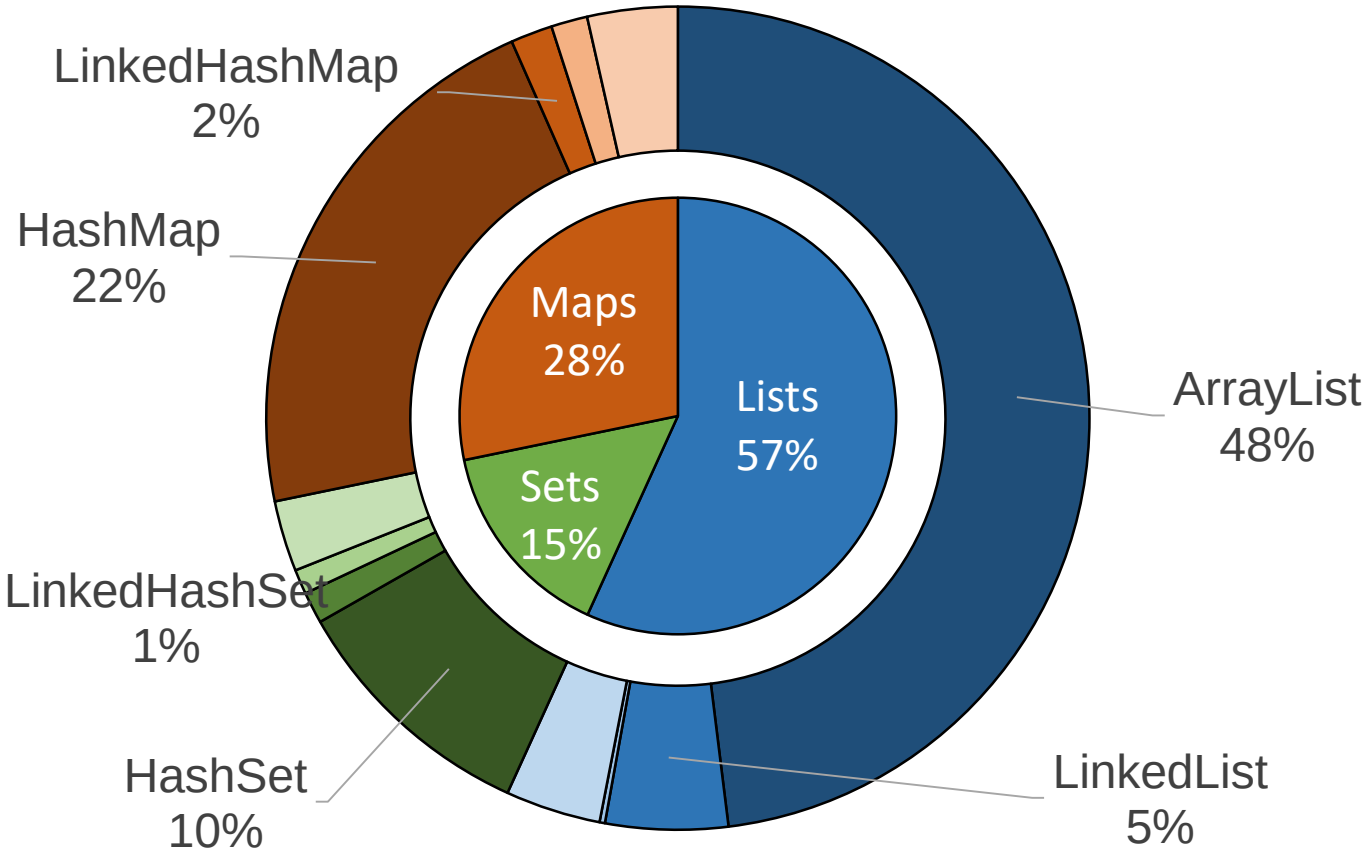
- Static Analysis
  - Use of Java Parser to extract variable declaration and allocation sites of Types with suffix:

    {List, Map, Set, Queue, Vector}
  - Manually removed false-positives

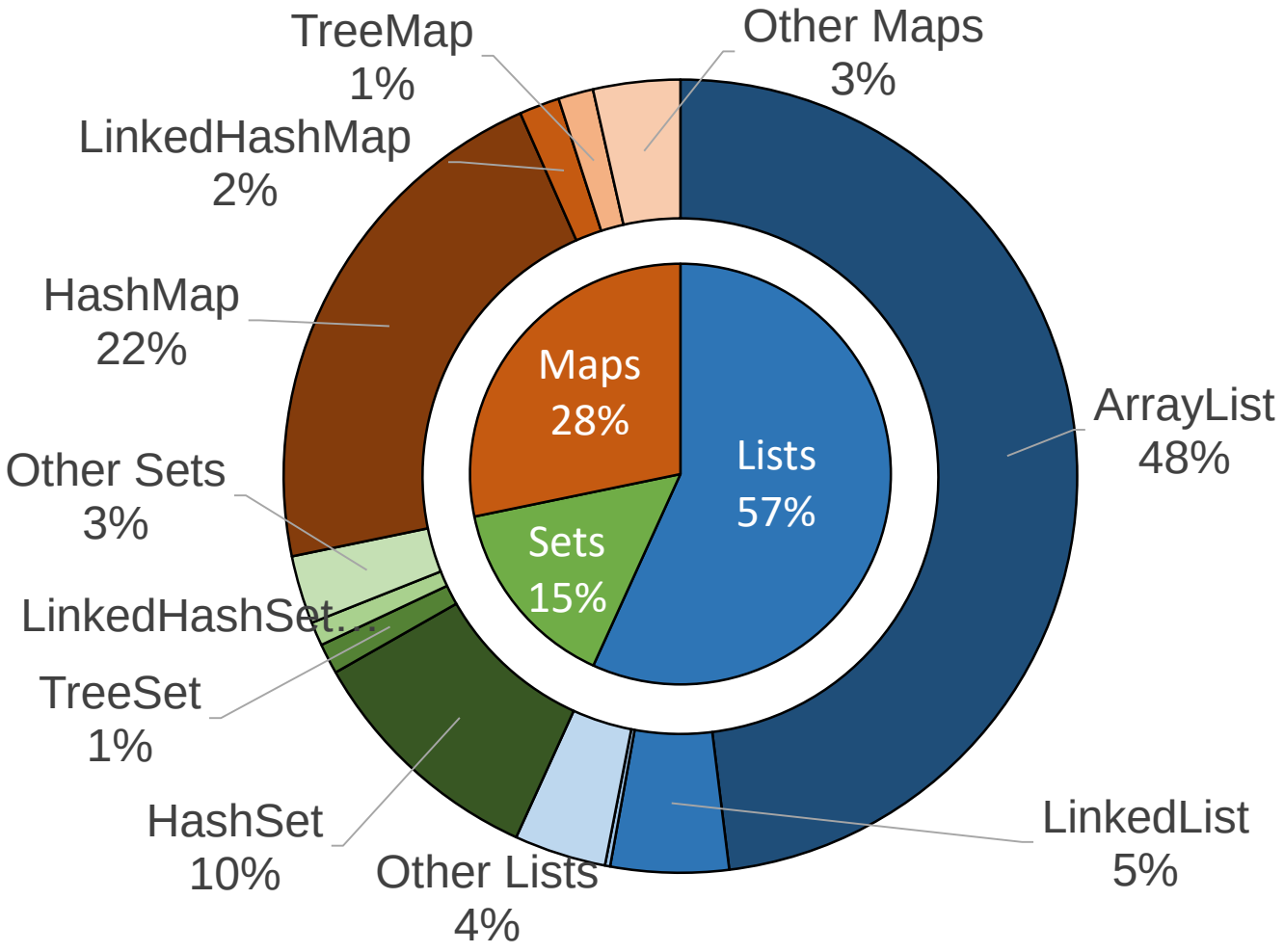
Developers Rarely Use non-Standard Collections



Developers Rarely Use non-Standard Collections



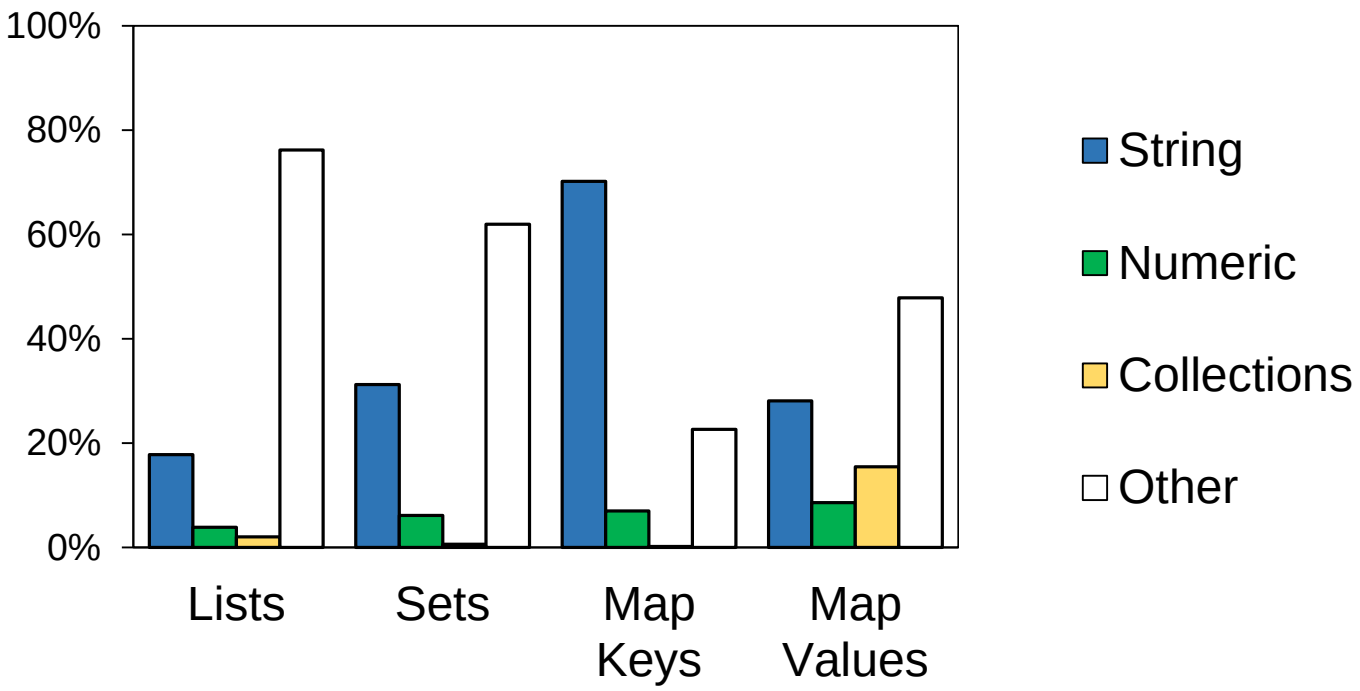
Developers Rarely Use non-Standard Collections



- Top 4 represent **86%** of all declared instantiations
- Non-Standard collections are declared **<4%**

Evaluate alternatives to ArrayList, HashMap, HashSet and LinkedList

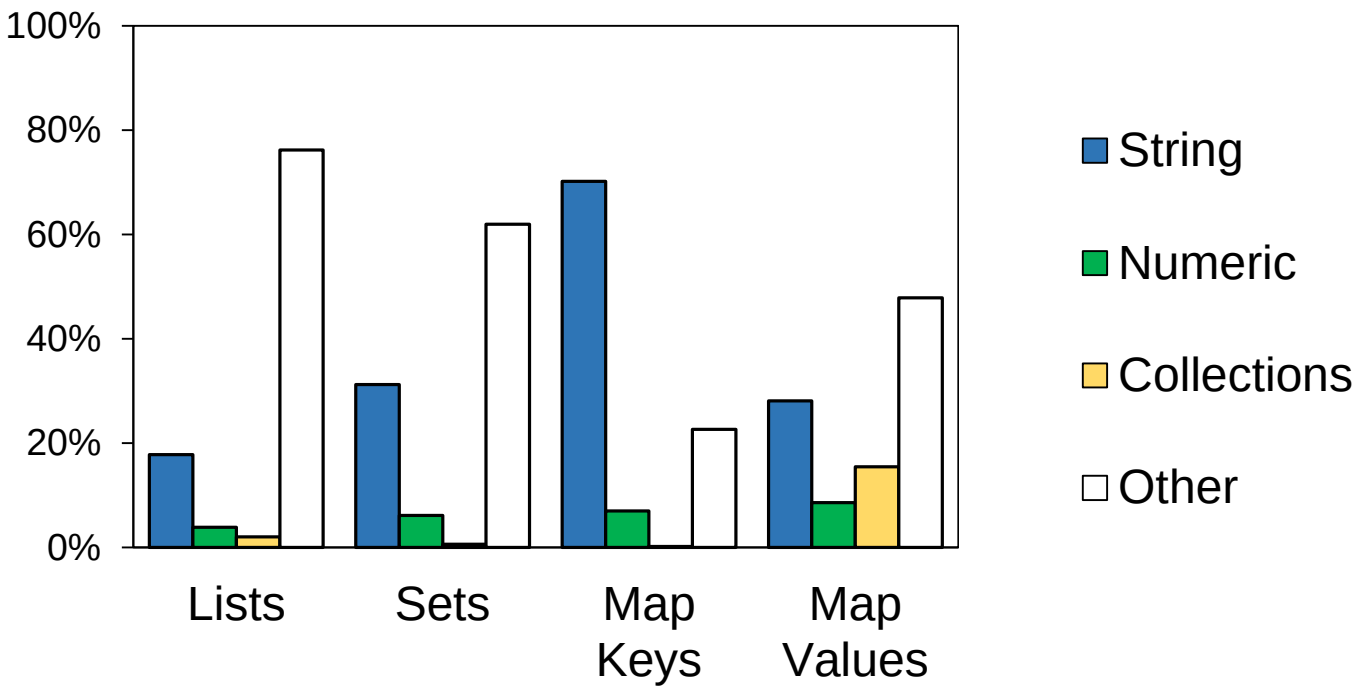
Commonly Used Element Data Types



From the categorized data types:

- Strings are the most commonly held data type, followed by Numeric

Commonly Used Element Data Types

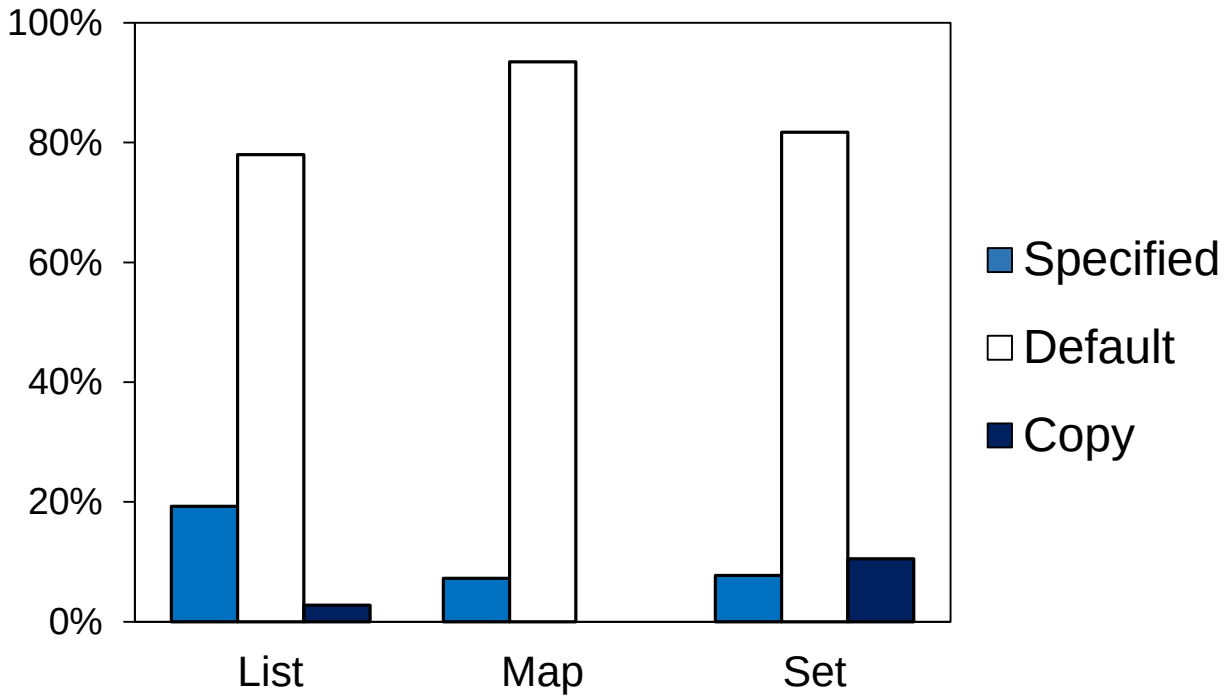


From the categorized data types:

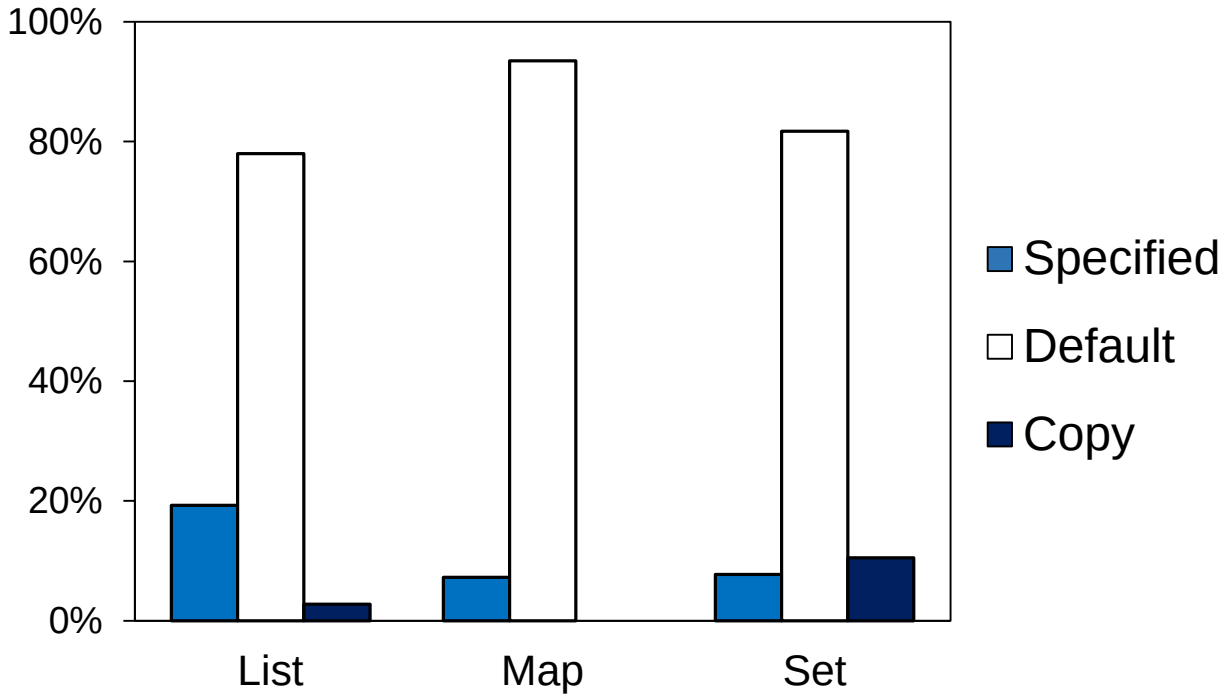
- Strings are the most commonly held data type, followed by Numeric

Evaluate collections holding Strings, Integer and Long

Initial Capacity is Rarely Specified



Initial Capacity is Rarely Specified



Evaluate collections with default Initial Capacity

Superior Alternatives

- **Superior Alternative:** a Non-Standard implementation that can outperform a Standard counterpart in terms of execution time and/or memory consumption.

- Can we find a superior alternative to the most commonly used collection types?

Experimental Study on Java Collections

- We selected 6 alternative libraries:
  - Repository Popularity (GitHub)
  - Appearance in previous partial benchmarks

Libraries	**Version**	**JCF Compatible**	**Available at**
Trove	3.0.3	yes	trove.starlight-systems.com
Guava	18.0	yes	github /google/guava
GSCollections	6.2.0	yes	/goldmansachs/gs-collections
HPPC	0.7.1	no	/carrotsearch/hppc
Fastutil	7.0.10	yes	/vigna/fastutil
Koloboke	0.6.8	yes	/leventov/Koloboke

Experimental Study on Java Collections

- Seven typical scenarios evaluated
  - populate, iterate, contains, get, add, remove, copy
- Collections holding from 100 to 1 million elements
- Alternatives to the most commonly used collections
  - JDK 1.8.0_65
  - ArrayList, HashMap, HashSet and LinkedList
    - Object collection alternatives
    - Primitive collection alternatives

CollectionsBench Suite

- We create a benchmark suite: **CollectionsBench**
  - Open Java Microbenchmark Harness

```
@Setup
public void setup() {
  fullList = this.createNewList();
  fullList.addAll(values); // Randomly
}                            generated

@Benchmark
public void iterate() {
  for (T element : fullList) {
    blackhole.consume(element);
   // Blackholes avoid dead-code
   // optimization
  }
}
```

CollectionsBench Suite

- We create a benchmark suite: **CollectionsBench**

Only the instantiation is needed for each collection type*

```
@Setup
public void setup() {
  fullList = this.createNewList();
  fullList.addAll(values); // Randomly
                           generated
}
```

```
return new ArrayList<T>();
return new LinkedList<T>();
return new FastList<T>();
```

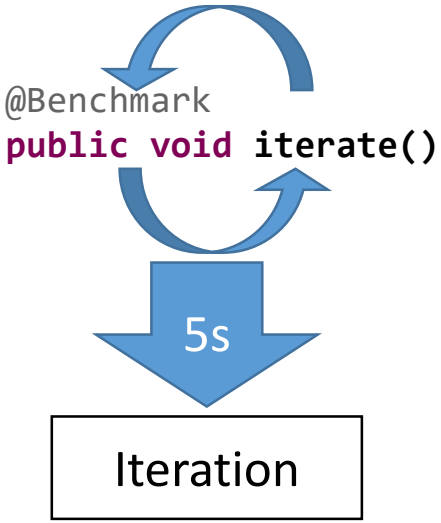
```
@Benchmark
public void iterate() {
  for (T element : fullList) {
    blackhole.consume(element);
   // Blackholes avoid dead-code
   // optimization
  }
}
```

Here we measure:

- Execution time (ns)
- Collection Overhead (allocation)
  - GC Profiler

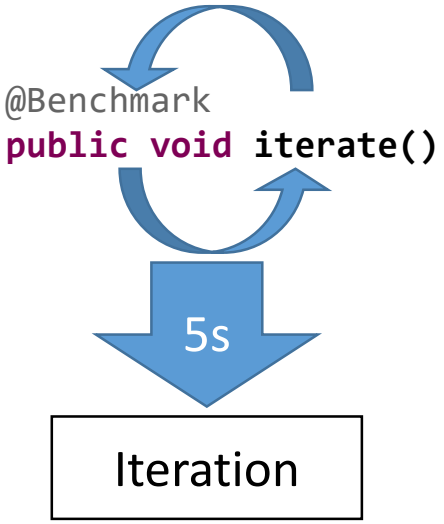
Experimental Planning

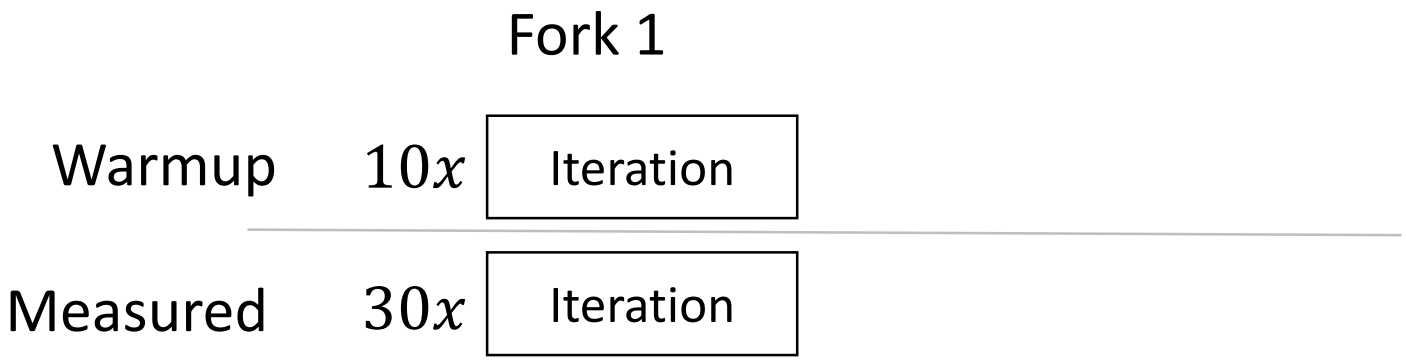
- To accomplish a **steady state** performance evaluation:



Experimental Planning

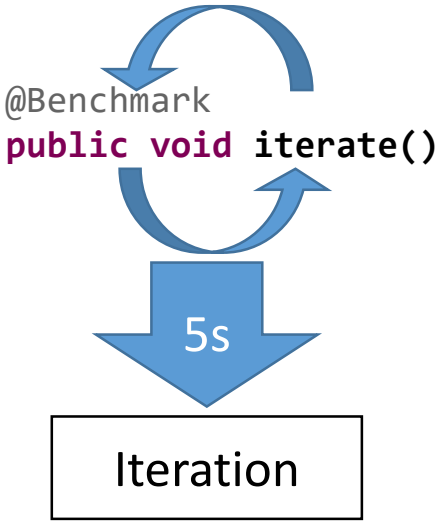
- To accomplish a **steady state** performance evaluation:





Experimental Planning

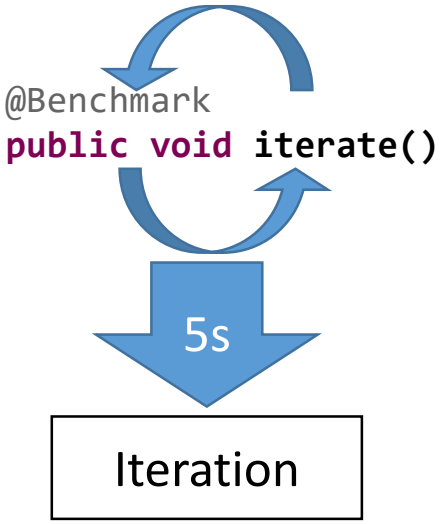
- To accomplish a **steady state** performance evaluation:



		Fork 1		Fork 2
Warmup	$10x$	Iteration	$10x$	Iteration
Measured	$30x$	Iteration	$30x$	Iteration

Experimental Planning

- To accomplish a **steady state** performance evaluation:



	Fork 1		Fork 2	
Warmup	$10x$	Iteration	$10x$	Iteration
Measured	$30x$	Iteration	$30x$	Iteration

AVG Time ± CI (99%)

AVG Memory allocated ± CI (99%)

Reporting Speedup/Slowdown

- We present the results of **alternatives** normalized to the **Standard** implementation performances.
  - Means with overlapping CI are set to **zero**

- We use the following *speedup*/*slowdown* definitions*:*

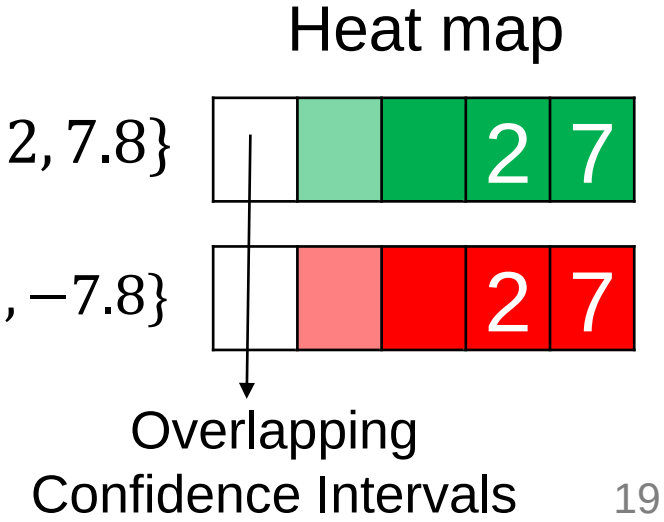
$$S = \begin{cases} \frac{T_{std}}{T_{alt}}, if \;\; T_{std} > T_{alt} \\ -\frac{T_{alt}}{T_{std}}, otherwise \end{cases}$$

Reporting Speedup/Slowdown

- We present the results of **alternatives** normalized to the **Standard** implementation performances.
  - Means with overlapping CI are set to **zero**

- We use the following *speedup*/*slowdown* definitions*:*

$$S = \begin{cases} \frac{T_{std}}{T_{alt}}, if \;\; T_{std} > T_{alt} \longrightarrow S = \{0, 1.5, 1.9, 2, 7.8\} \\ -\frac{T_{alt}}{T_{std}}, otherwise \longrightarrow S = \{0, -1.5, -1.9, -2, -7.8\} \end{cases}$$



Reporting Memory Overhead

- For the memory comparison we present the collection overhead reduction **per element** (with compressed object pointers)

Reporting Memory Overhead

- For the memory comparison we present the collection overhead reduction **per element** (with compressed object pointers)

  For instance

  Collection X: **100** bytes per element
  Collection Y: **10** bytes per element

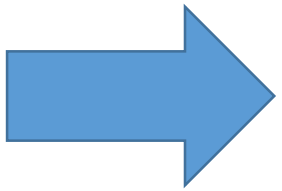
- Evaluated on the **copy** scenario

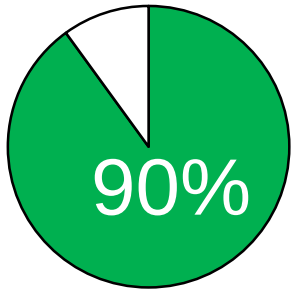
Reporting Memory Overhead

- For the memory comparison we present the collection overhead reduction **per element** (with compressed object pointers)

For instance

Collection X: **100** bytes per element
Collection Y: **10** bytes per element



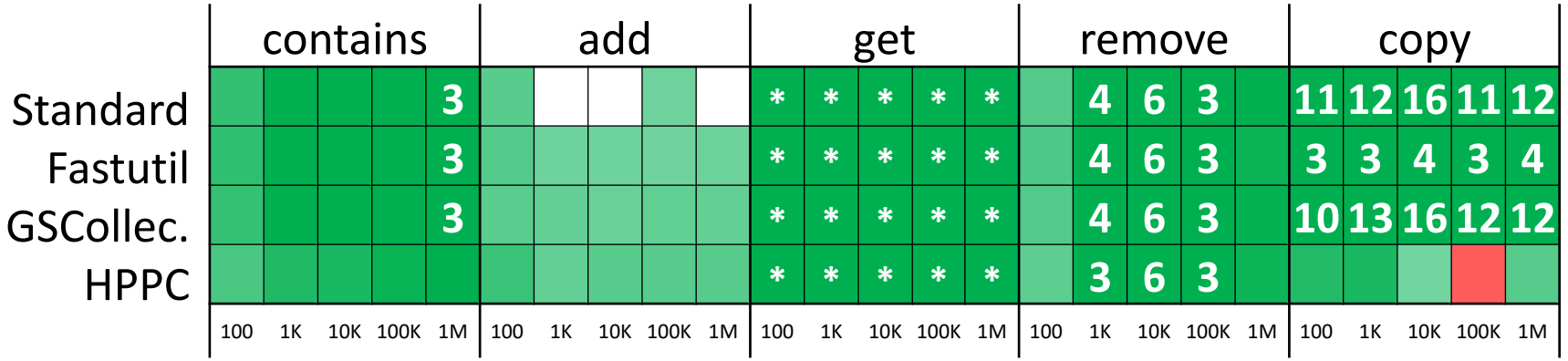


Reduction of Collection Overhead

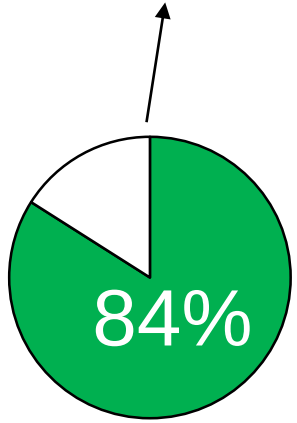
- Evaluated on the **copy** scenario

Superior Alternatives: LinkedList

- LinkedList was outperformed by **every** ArrayList alternative



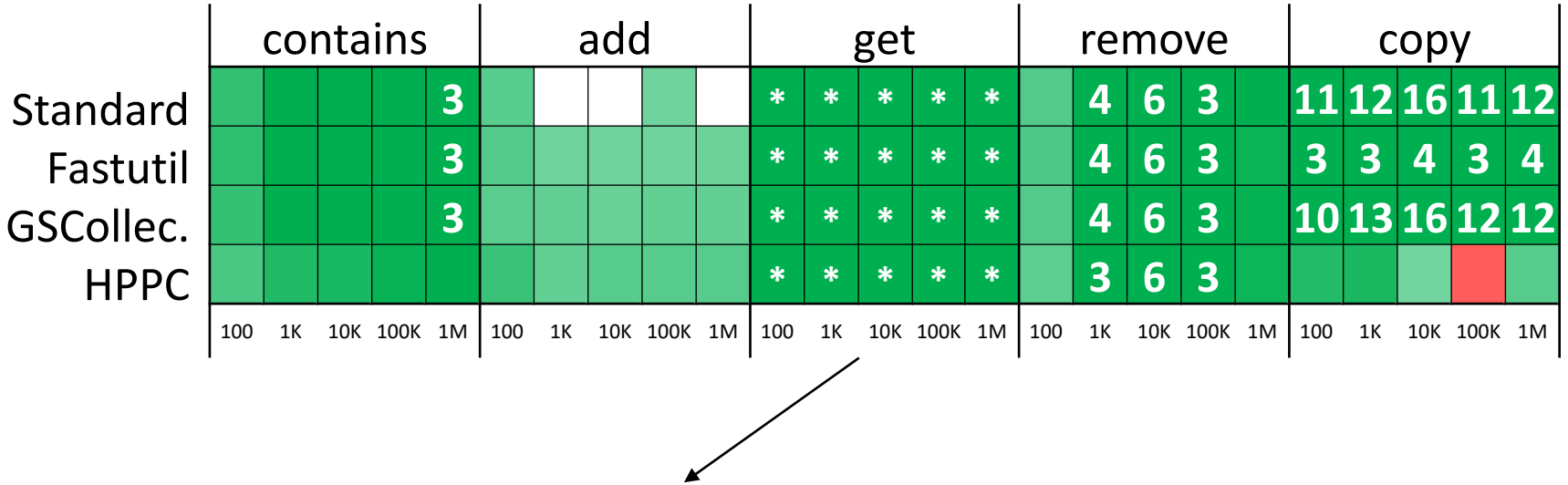
JCF LinkedList$Entry consumes **24 bytes**



Reduction of Collection Overhead

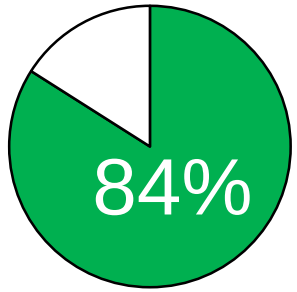
Superior Alternatives: LinkedList

- LinkedList was outperformed by **every** ArrayList alternative



Asymptotic disadvantage

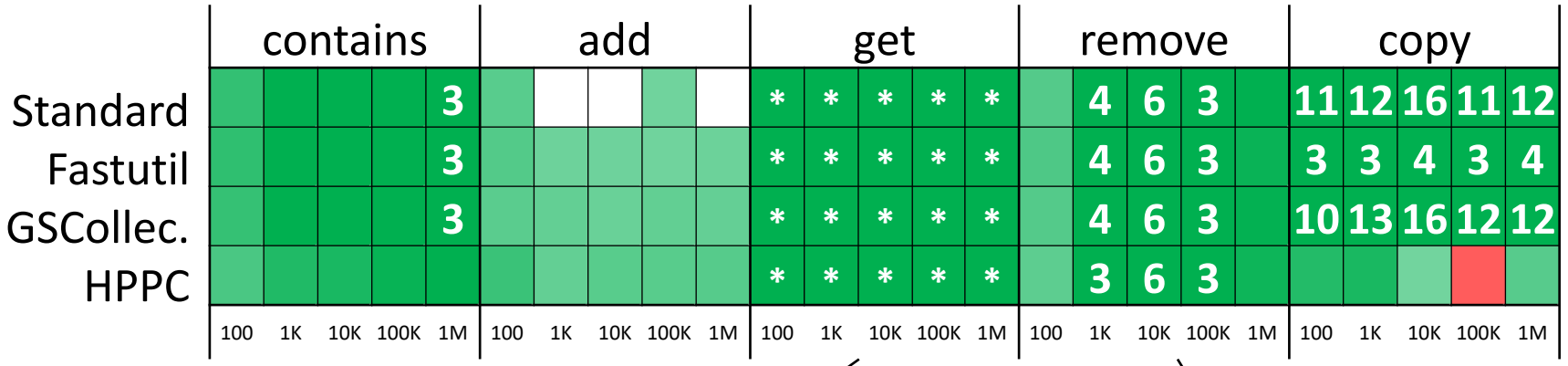
JCF LinkedList$Entry consumes **24 bytes**



Reduction of Collection Overhead

Superior Alternatives: LinkedList

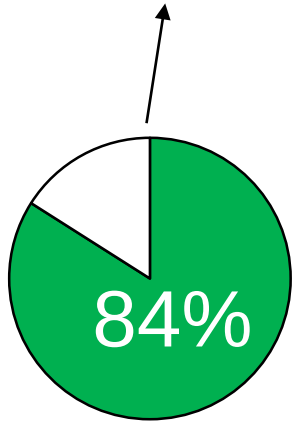
- LinkedList was outperformed by **every** ArrayList alternative



Asymptotic disadvantage

Asymptotic advantage

JCF LinkedList$Entry consumes **24 bytes**

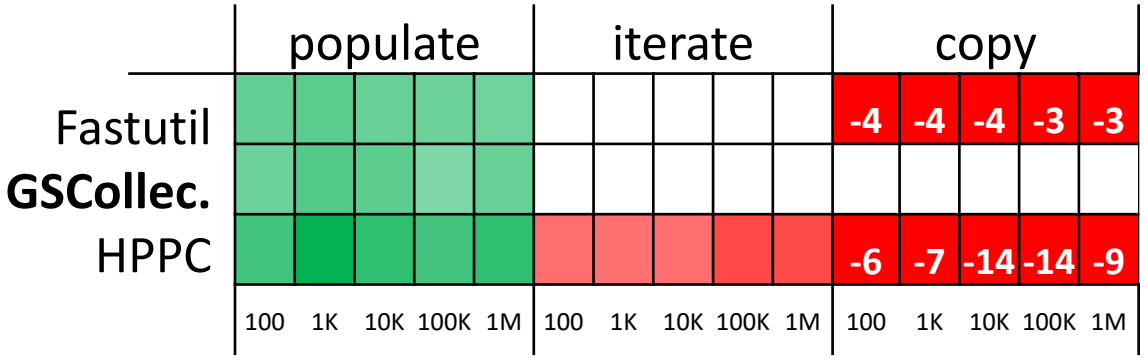


Reduction of Collection Overhead

```
public boolean remove(Object o)
```

Superior Alternatives: ArrayList

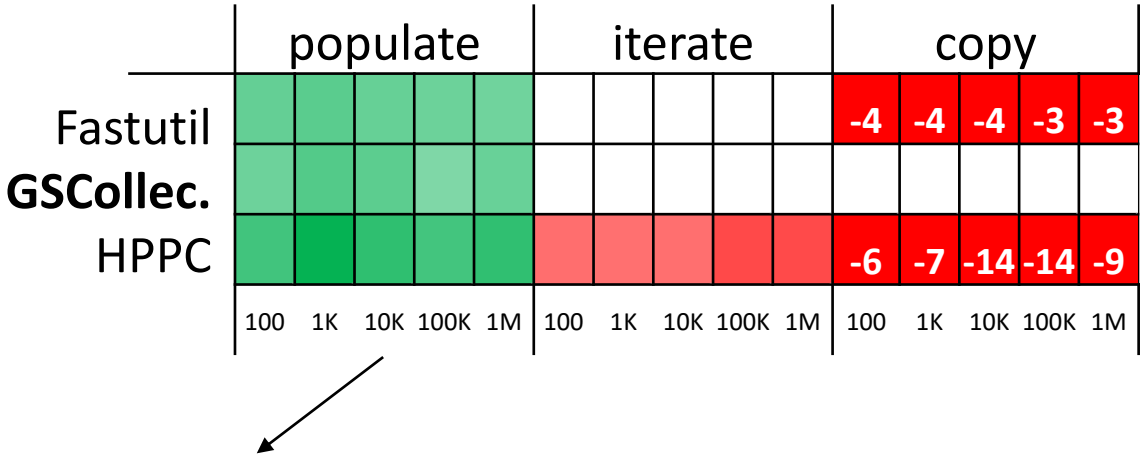
- *GSCollections* provides a superior alternative
  - Faster when populating the list (no time penalty)



No memory difference

Superior Alternatives: ArrayList

- *GSCollections* provides a superior alternative
  - Faster when populating the list (no time penalty)



No memory difference

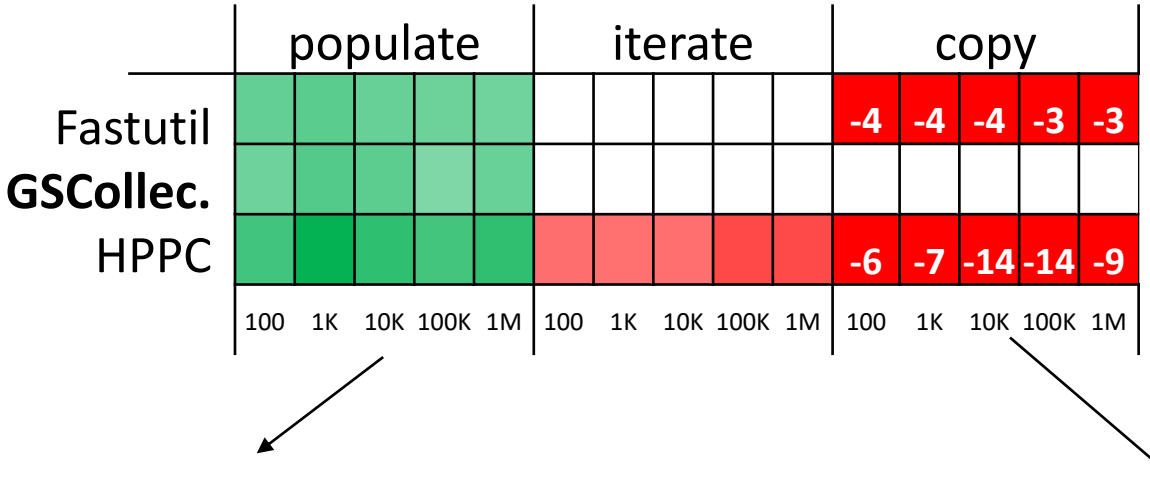
Distinct Array copy calls

Std: `Arrays.copyOf();`

Alt: `System.arraycopy();`

Superior Alternatives: ArrayList

- *GSCollections* provides a superior alternative
  - Faster when populating the list (no time penalty)



No memory difference

Distinct Array copy calls

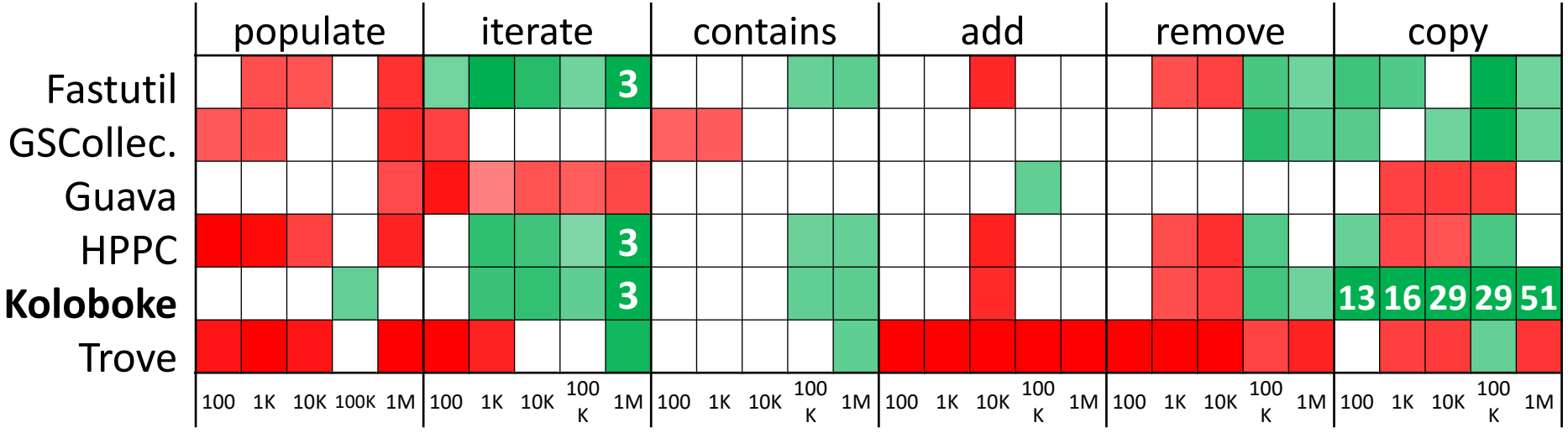
Std: `Arrays.copyOf();`

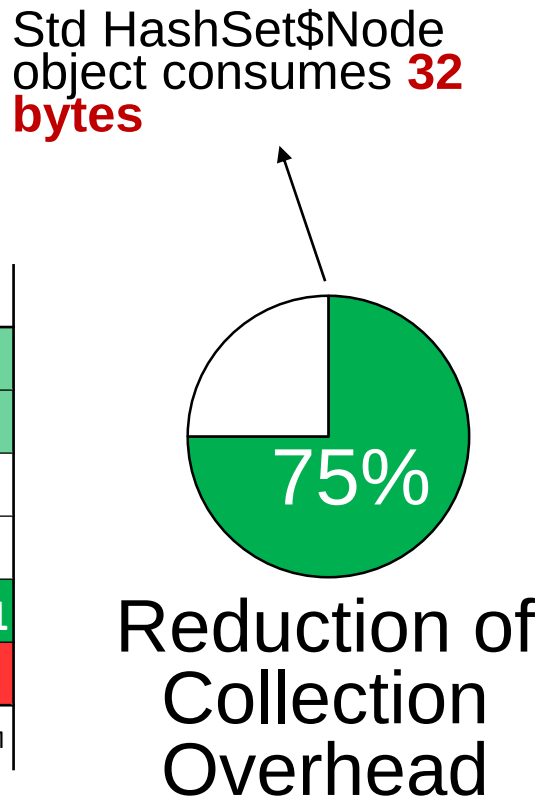
Alt: `System.arraycopy();`

HPPC adds each element instead of copying the array

Superior Alternatives: HashSet

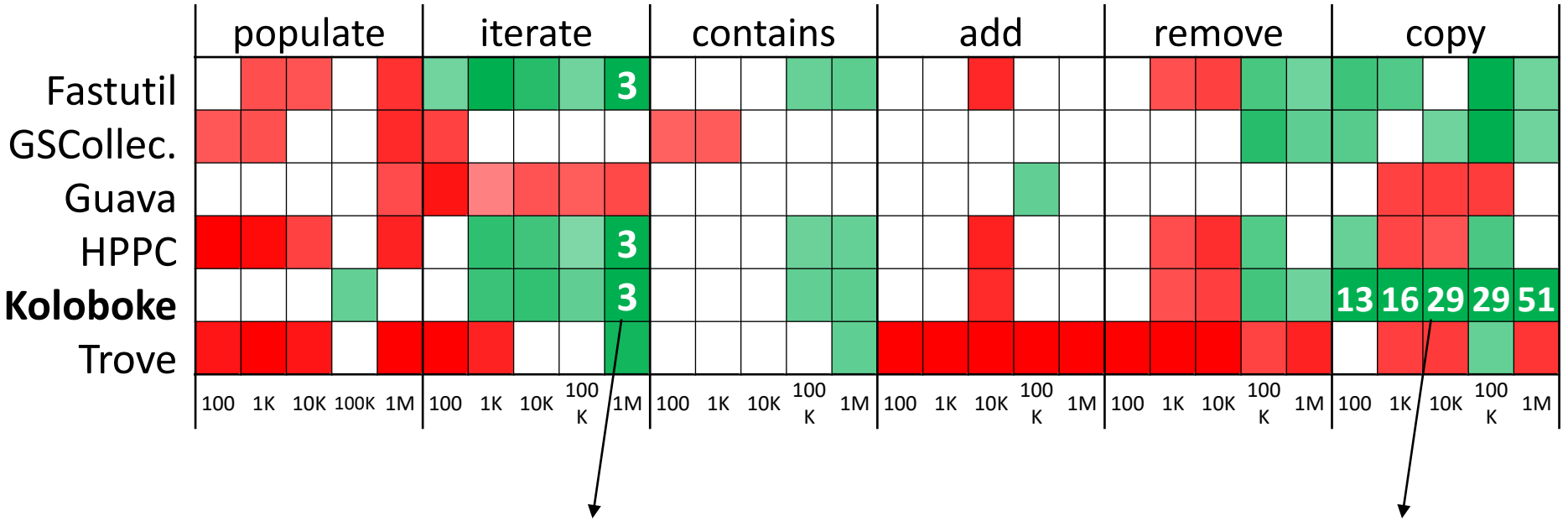
- Koloboke provides a superior alternative
  - Fastutil is a solid 2nd option





Superior Alternatives: HashSet

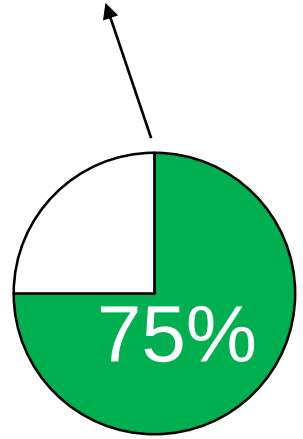
- Koloboke provides a superior alternative
  - Fastutil is a solid 2nd option



Impact of memory efficiency on time

Koloboke performs a memory copy of its HashTable

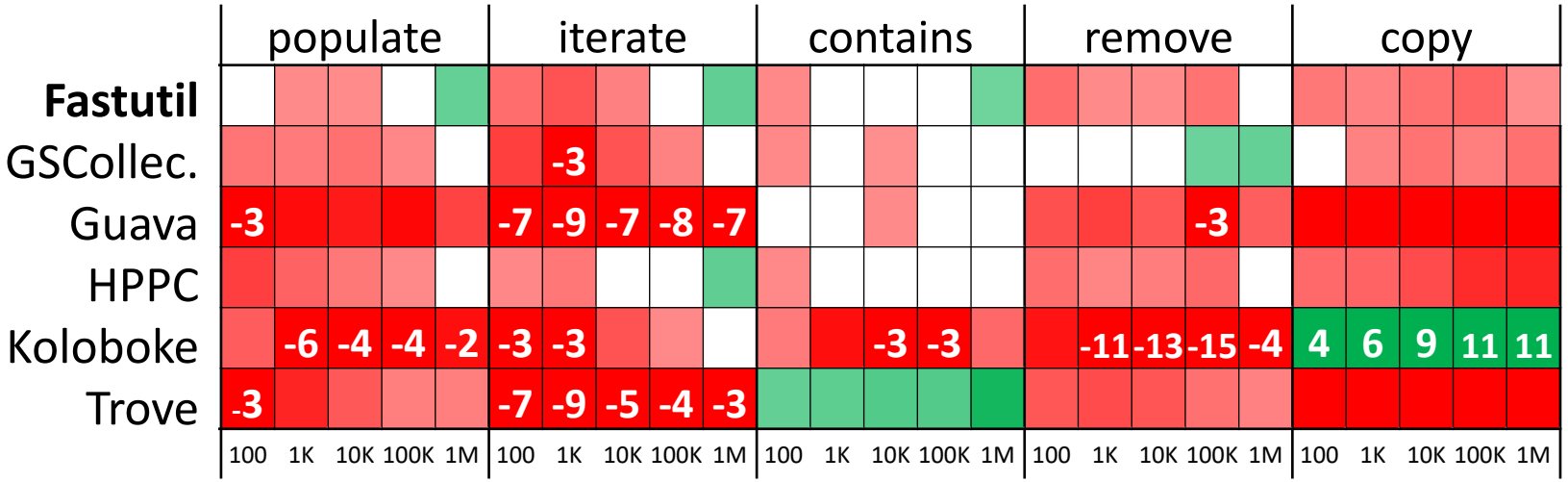
Std HashSet$Node object consumes **32 bytes**



Reduction of Collection Overhead

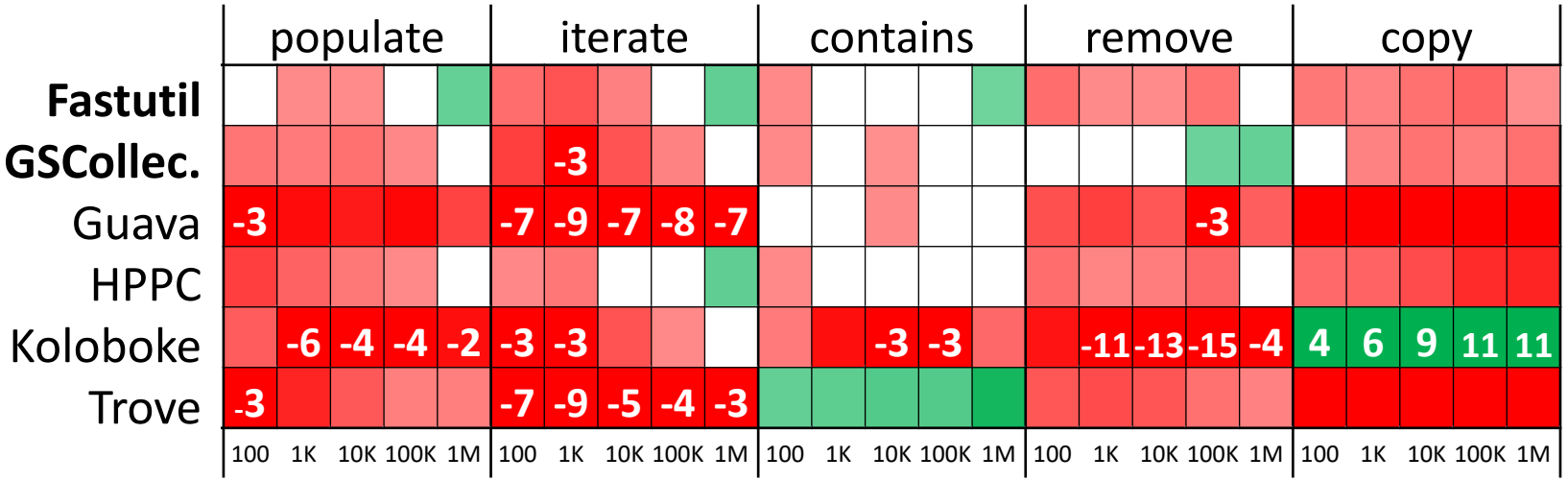
Superior Alternatives: HashMap

- Standard HashMap is a **solid** implementation
  - No superior alternatives on time

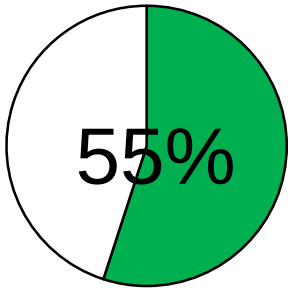


Superior Alternatives: HashMap

- Standard HashMap is a **solid** implementation
  - No superior alternatives on time



- Fastutil provides a superior alternative on memory consumption



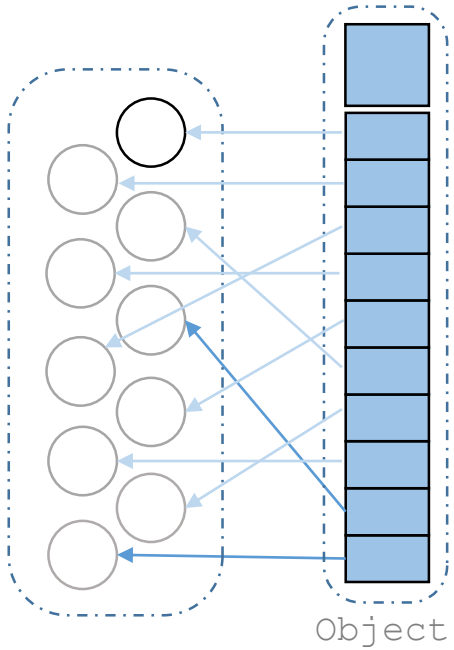
Reduction of Collection Overhead

Std. HashMap$Node object consumes **32 bytes**

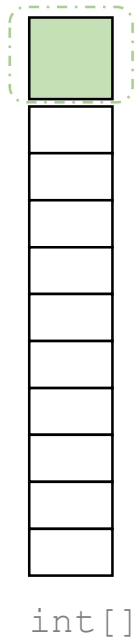
Object vs Primitive Collections

- Reducing collection footprint: overhead + element footprint

Object collection
ArrayList<>

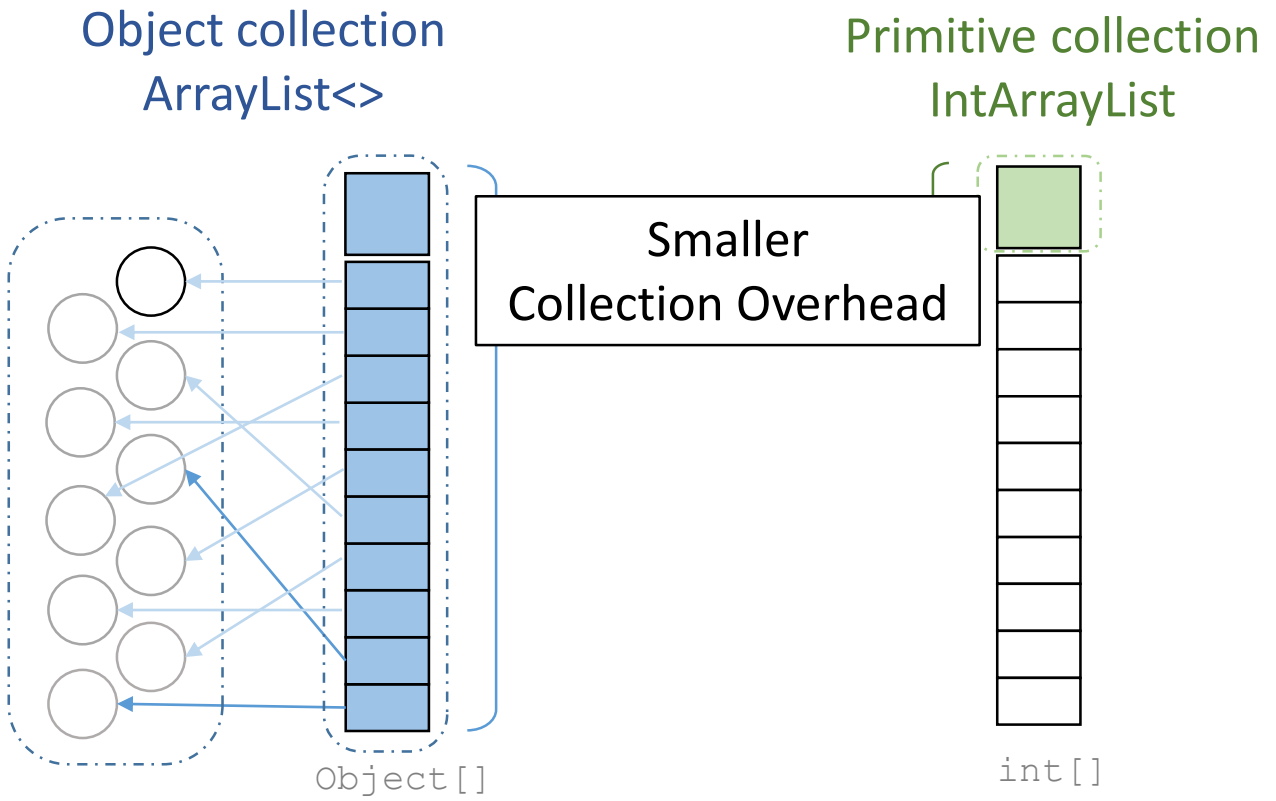


Primitive collection
IntArrayList



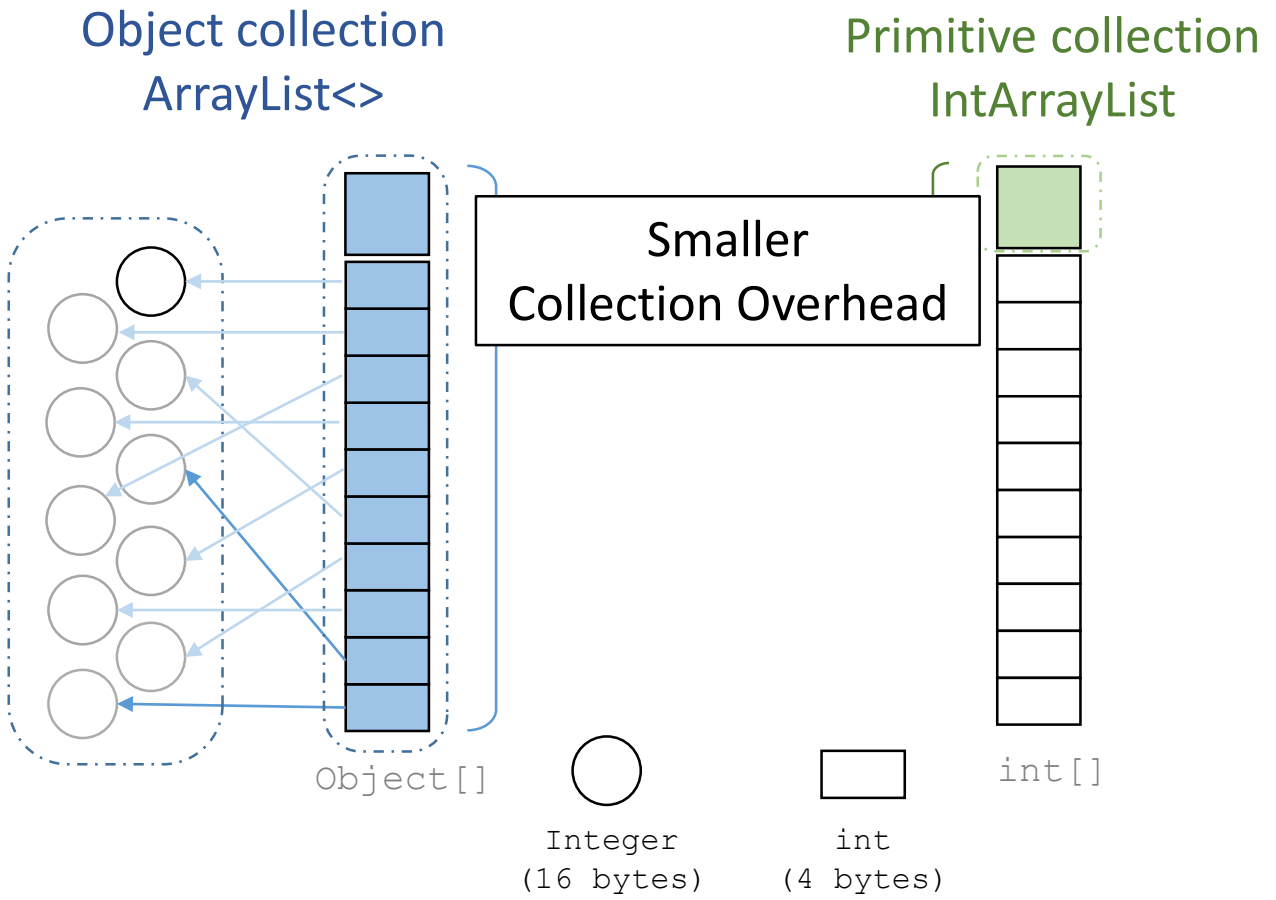
Object vs Primitive Collections

- Reducing collection footprint: overhead + element footprint



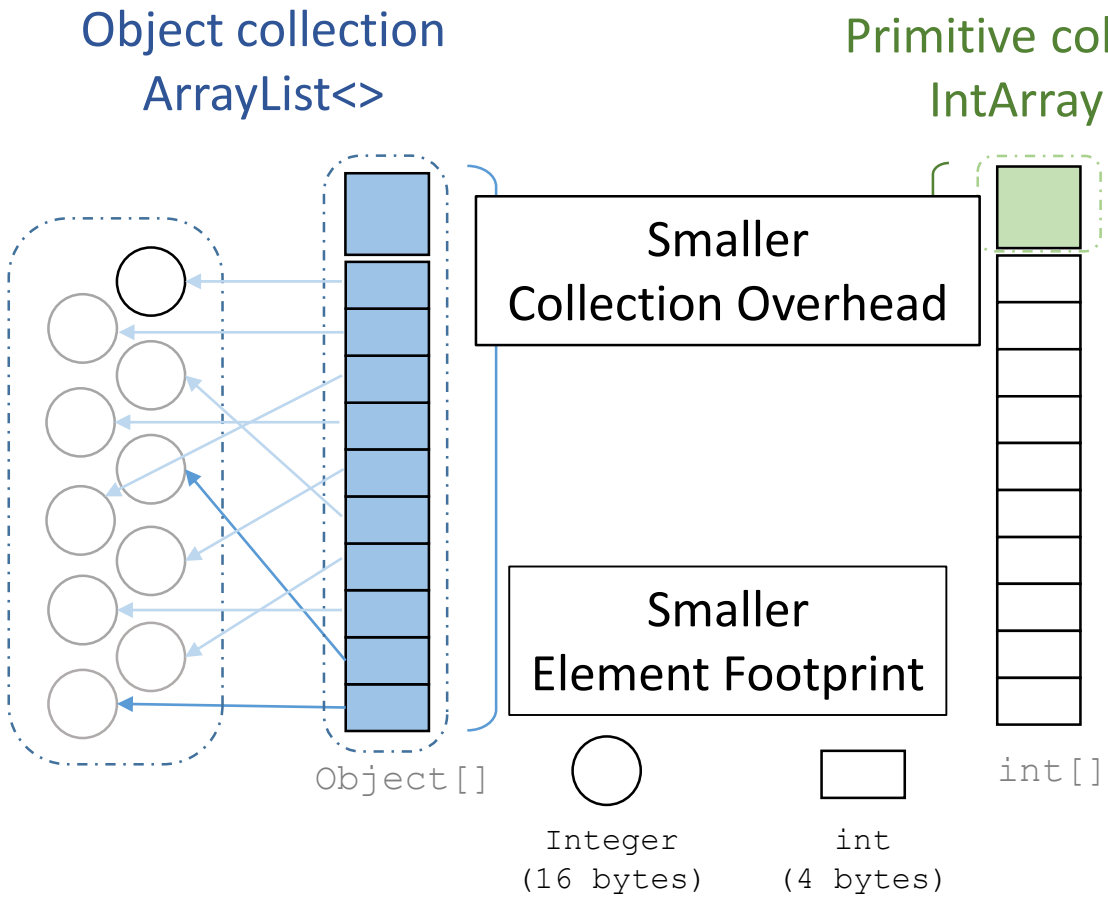
Object vs Primitive Collections

- Reducing collection footprint: overhead + element footprint



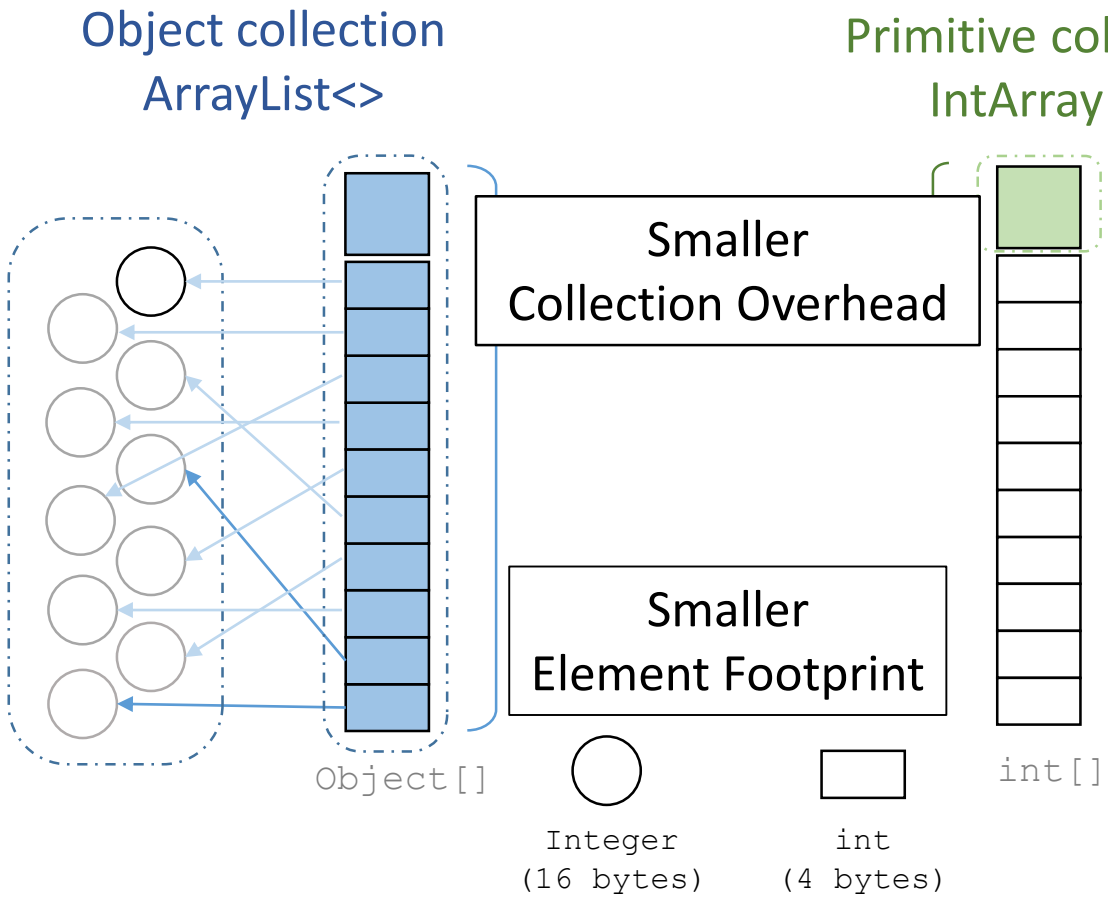
Object vs Primitive Collections

- Reducing collection footprint: overhead + element footprint



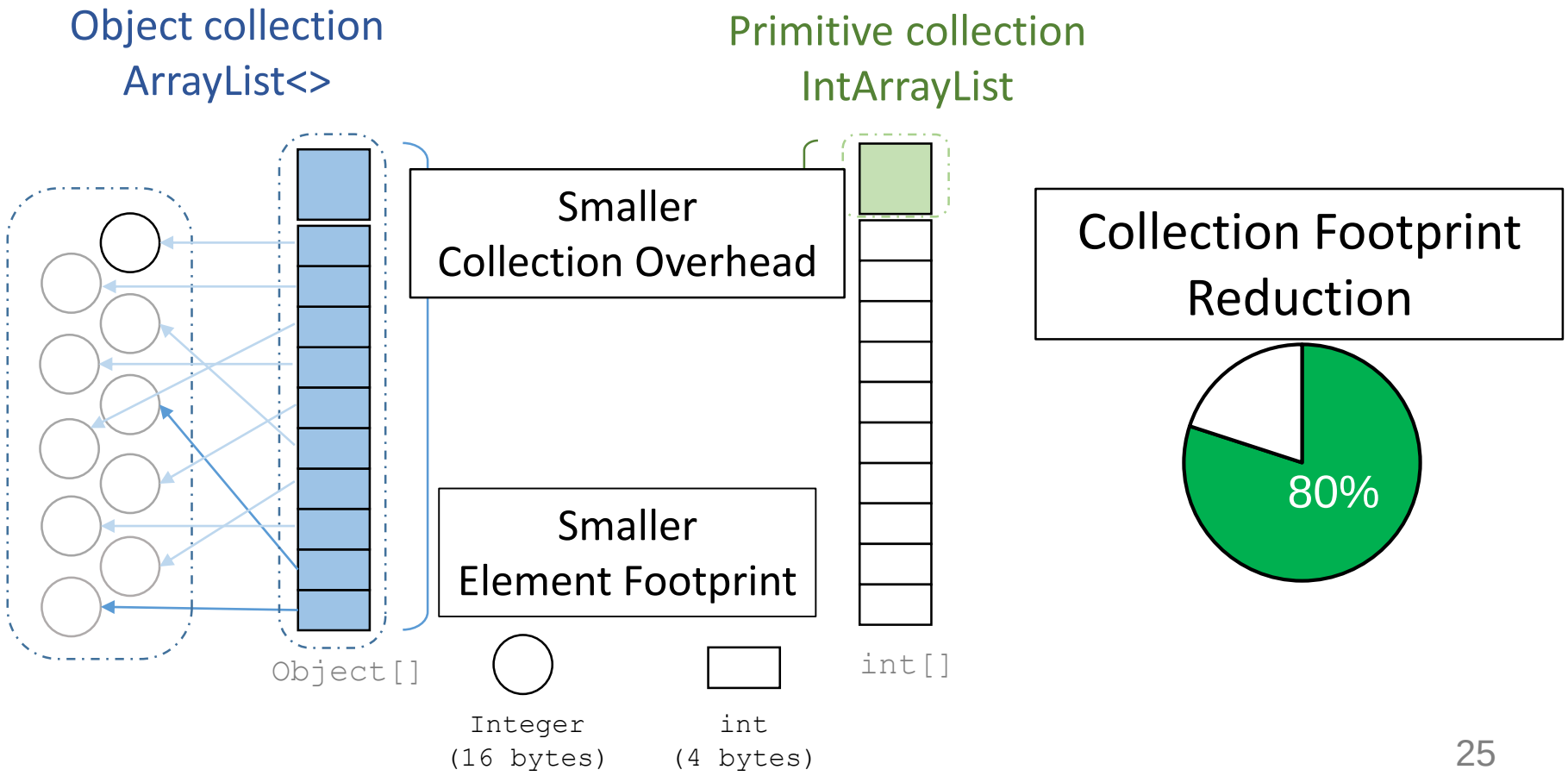
Object vs Primitive Collections

- Reducing collection footprint: overhead + element footprint



Object vs Primitive Collections

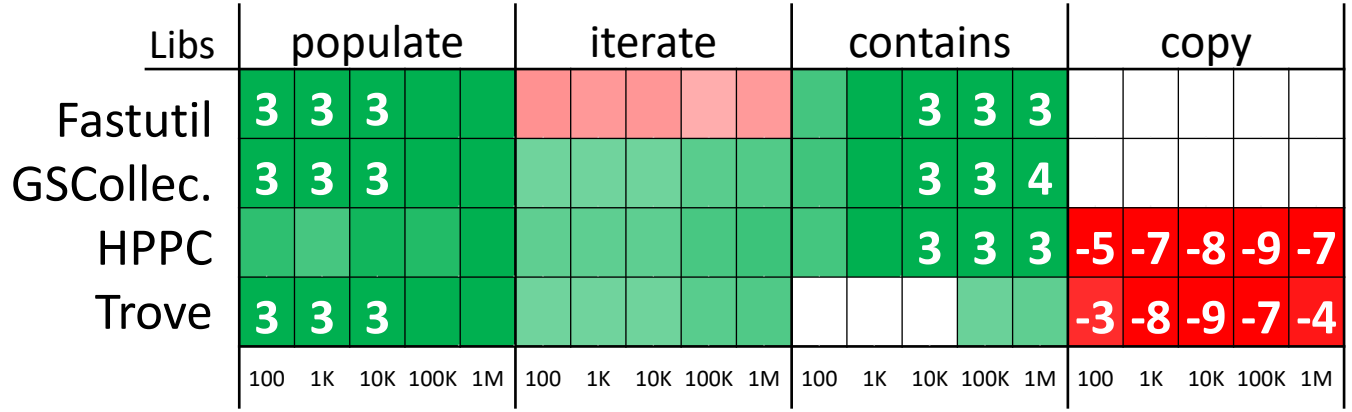
- Reducing collection footprint: overhead + element footprint



Superior Alternatives: Primitive Collections

- We found superior alternatives to all three abstraction types
  - Data Type: Integer to int
- Performance varies considerably from distinct libraries for multiple reasons

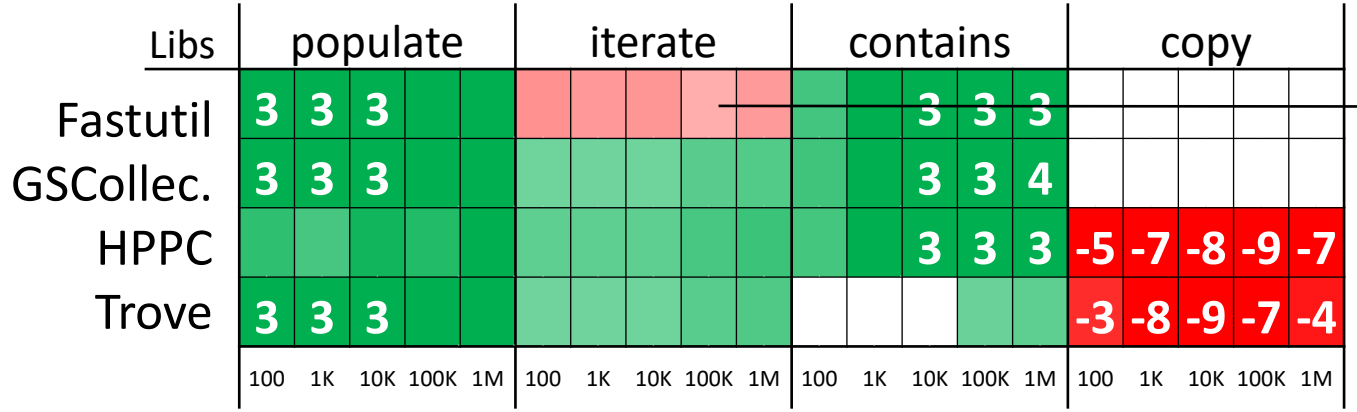
For instance, ArrayList primitive implementations



Superior Alternatives: Primitive Collections

- We found superior alternatives to all three abstraction types
  - Data Type: Integer to int
- Performance varies considerably from distinct libraries for multiple reasons

For instance, ArrayList primitive implementations



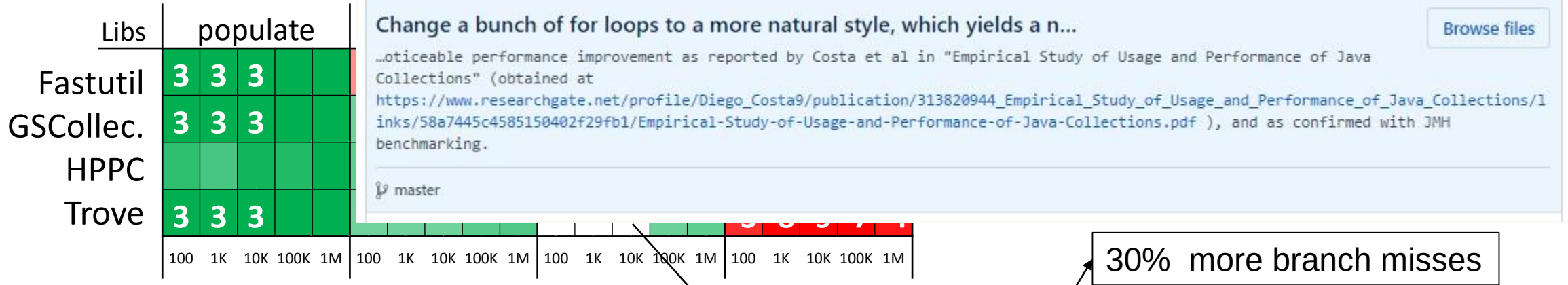
`for(Object..)`

5x slower than `for(IntProcedure)` commonly implemented

Superior Alternatives: Primitive Collections

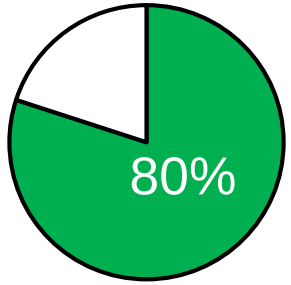
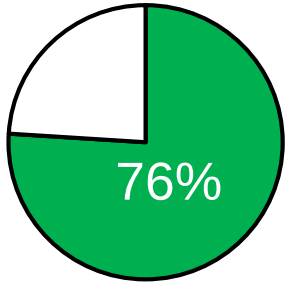
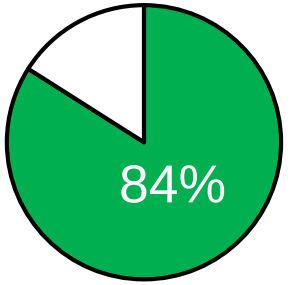
- We found superior alternatives to all three abstraction types
  - Data Type: Integer to int
- Performance varies considerably from distinct libraries for multiple reasons

For instance, ArrayList primitive implementations



Superior Alternative: Primitive Collections

- GSCollections, Koloboke and Fastutil provide solid superior alternatives

	ArrayList	HashMap	HashSet
Footprint Reduction `Integer` → `int`			
Time impact (1M elements)	+2x populate +4x contains +2x remove	+2x populate -2x iterate +6x/+4x copy	+4x populate +4x/+3x iterate +33x/+7x copy

Summary

- There are performance opportunities on Alternative Collection Frameworks
  - Time/memory improvement with moderate refactor effort
- We provide a Guideline to assist developers on:
  - Identifying superior alternative implementations
  - Which scenarios an alternative could lead to a substantial performance improvement
- CollectionsBench is open-source and available at GitLab

Thank you for your time

Questions?

diego.costa@informatik-uni.heidelberg.de