JBrainy: Micro-benchmarking Java Collections with Interference

WORK IN PROGRESS PAPER

Noric Couderc   Emma Söderberg   Christoph Reichenbach

Collection Selection Problem

Collection Selection
Micro-Benchmarking
Results
Conclusion

The Problem

Here is Java code:

```
Set countries = new _______; // What should we write here?
s.add("France");
s.add("Sweden");
s.add("Germany");
// ...
s.contains("Norway");
// ...
s.clear();
```

The answer depends of **the context**: How will we use our collection?

Can we use the most popular?

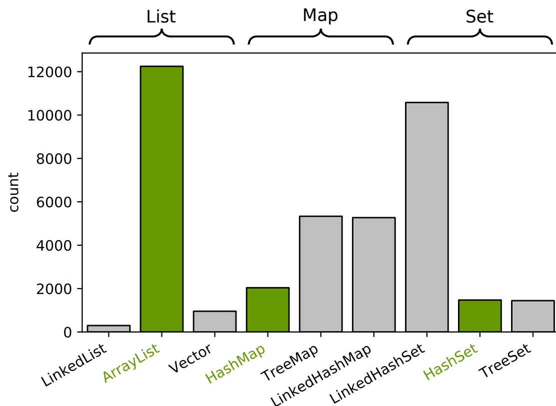


Figure: Distribution of fastest benchmarks

Can we use the most popular?

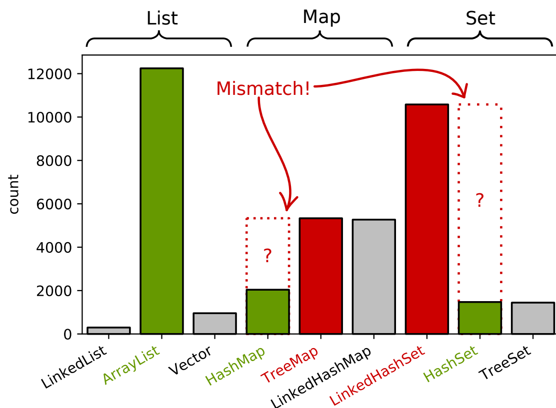


Figure: Distribution of fastest benchmarks

LUND UNIVERSITY

Collection Selection Problem

The Problem

Here is Java code:

```
Set countries = new _______; // What should we write here?
s.add("France");
s.add("Sweden");
s.add("Germany");
// ...
s.contains("Norway");
// ...
s.clear();
```

To pick the right implementation class you need to know:

- How you will use your collection: It's **usage profile**
- Each implementation class' strengths and weaknesses: It's **cost profile**

LUND UNIVERSITY

Usage Profiles

Collection Selection
Micro-Benchmarking
Results
Conclusion

Synthesized usage profile

method: clear(), equals(Object), add(Object), removeAll(Collection), size(), hashCode(), retainAll(Collection), iterator(), toArray(), contains(Object), toArray(Object[]), containsAll(Collection), isEmpty(), remove(Object), addAll(Collection)

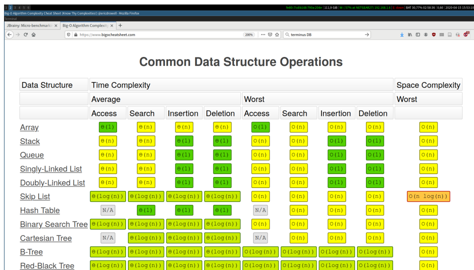
number of calls: 0, 100, 200, 300, 400, 500

LUND UNIVERSITY

Cost Profiles

Collection Selection
Micro-Benchmarking
Results
Conclusion

Usually, developers use tables such as this one:



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Figure: Big-O Cheatsheet by Eric Rowell

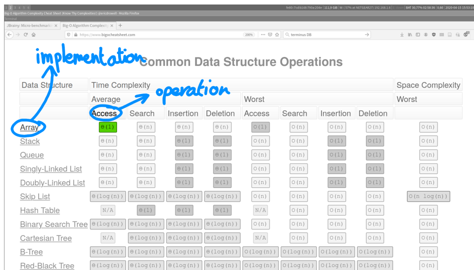
These can be derived theoretically or empirically.

LUND UNIVERSITY

Cost Profiles

Collection Selection
Micro-Benchmarking
Results
Conclusion

Usually, developers use tables such as this one:



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Cartesian Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
B-Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Red-Black Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$

Figure: Big-O Cheatsheet by Eric Rowell

These can be derived theoretically or empirically.

LUND UNIVERSITY

Cost Profiles

Usually, developers use tables such as this one:

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Cartesian Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
B-Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Red-Black Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$

cost

Figure: Big-O Cheatsheet by Eric Rowell

These can be derived theoretically or empirically.

Interference

Collection Selection
Micro-Benchmarking
Results
Conclusion

LUND UNIVERSITY

Assumptions

Maybe we need to consider **interference** between methods called?

$$\texttt{ArrayList} + [\text{get}, \text{add}, \text{get}, \ldots] = \bigstar\bigstar\bigstar$$
$$\texttt{LinkedList} + [\text{get}, \text{add}, \text{get}, \ldots] = \bigstar$$

Our solution: JBrainy's Microbenchmarking

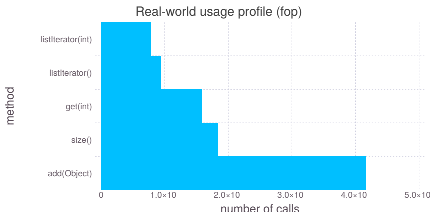
Question

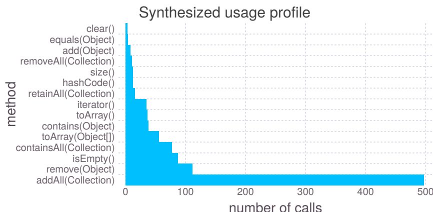
How do we benchmark if there is interference?

Answer

Synthetic benchmarking, with:

- A **uniform** usage profile (Jung et al's Brainy).
- A **more realistic** usage profile (Our tool, JBrainy).





Hardware

Collection Selection
Micro-Benchmarking
Results
Conclusion

- Intel(R) Core(TM) i7-3820 CPU @ 3.60GHz
- 16 GB of RAM
- Ubuntu 18.04 (Linux 4.18.0-15-generic),
- OpenJDK 10.0.2.

LUND UNIVERSITY

Results: Polyvalence

Collection Selection
Micro-Benchmarking
Results
Conclusion

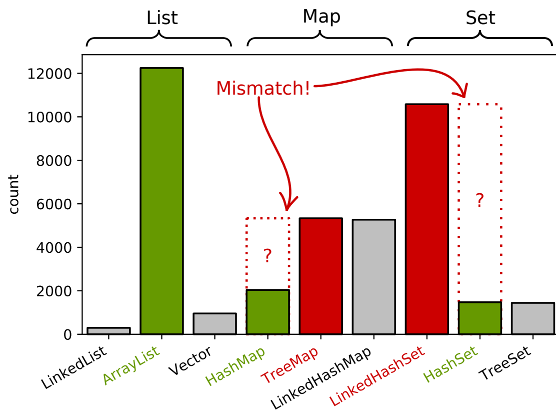


Figure: Distribution of fastest benchmarks

LUND UNIVERSITY

Results: Food for Thought

Lists compared to ArrayList

ArrayList is faster across the board.

- LinkedList: **0.5**
- Vector: **0.9**

Sets compared to HashSet

- LinkedHashSet usually faster: particularly:
  - `toArray()`: **2.96**
  - `toArray(Object [])`: **2.85**
  - `add()`: **2.10**
- TreeSet:
  - `clear()`: **1.18**

Maps compared to HashMap

- LinkedHashMap usually faster, particularly:
  - `put()`: **1.28**
  - `hashCode()`: **1.20**
  - `remove()`: **1.10**
- TreeMap:
  - `clear()`: **1.07**

Legend

`method()`: Implicit traversal

3.0: Speedup

Why does this matter?

Collection Selection
Micro-Benchmarking
Results
Conclusion

Our results indicate that **popular** collections are often not the optimal choice, probably because `LinkedHashMap` and `LinkedHashSet` exploit **interference**.

Conclusion

Collection Selection
Micro-Benchmarking
Results
Conclusion

- We discussed the issues at play when optimizing collection use.
- We introduced our method of micro-benchmarking collections.
- Our results indicate that:
  - The most **popular** collections do **not** necessarily yield the best performance.
  - Interference plays a role in collection performance.
- Future work:
  - Investigate the reasons for speedups
  - Try with more JVMs and more hardware
  - Try it on real programs!

LUND
UNIVERSITY