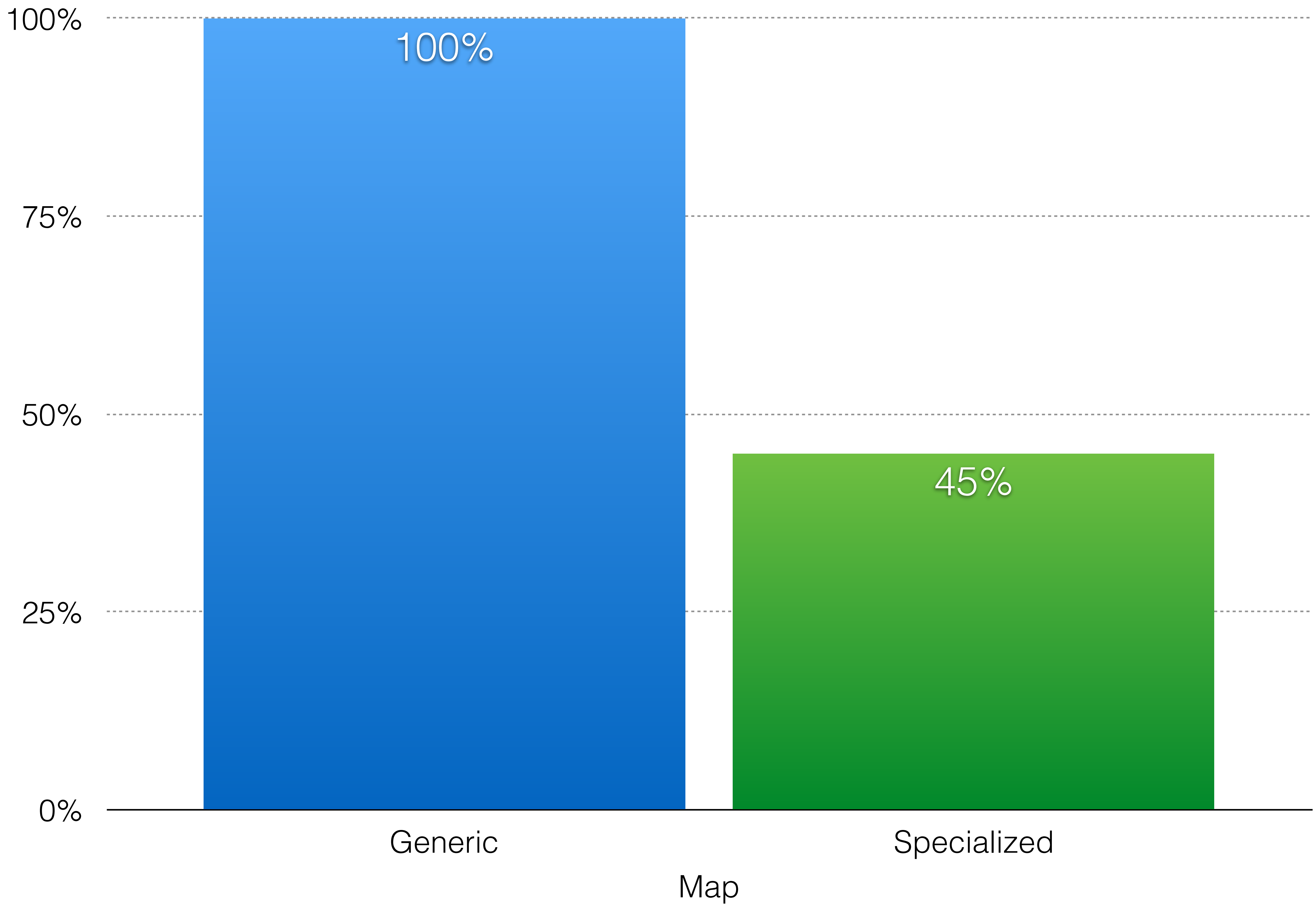
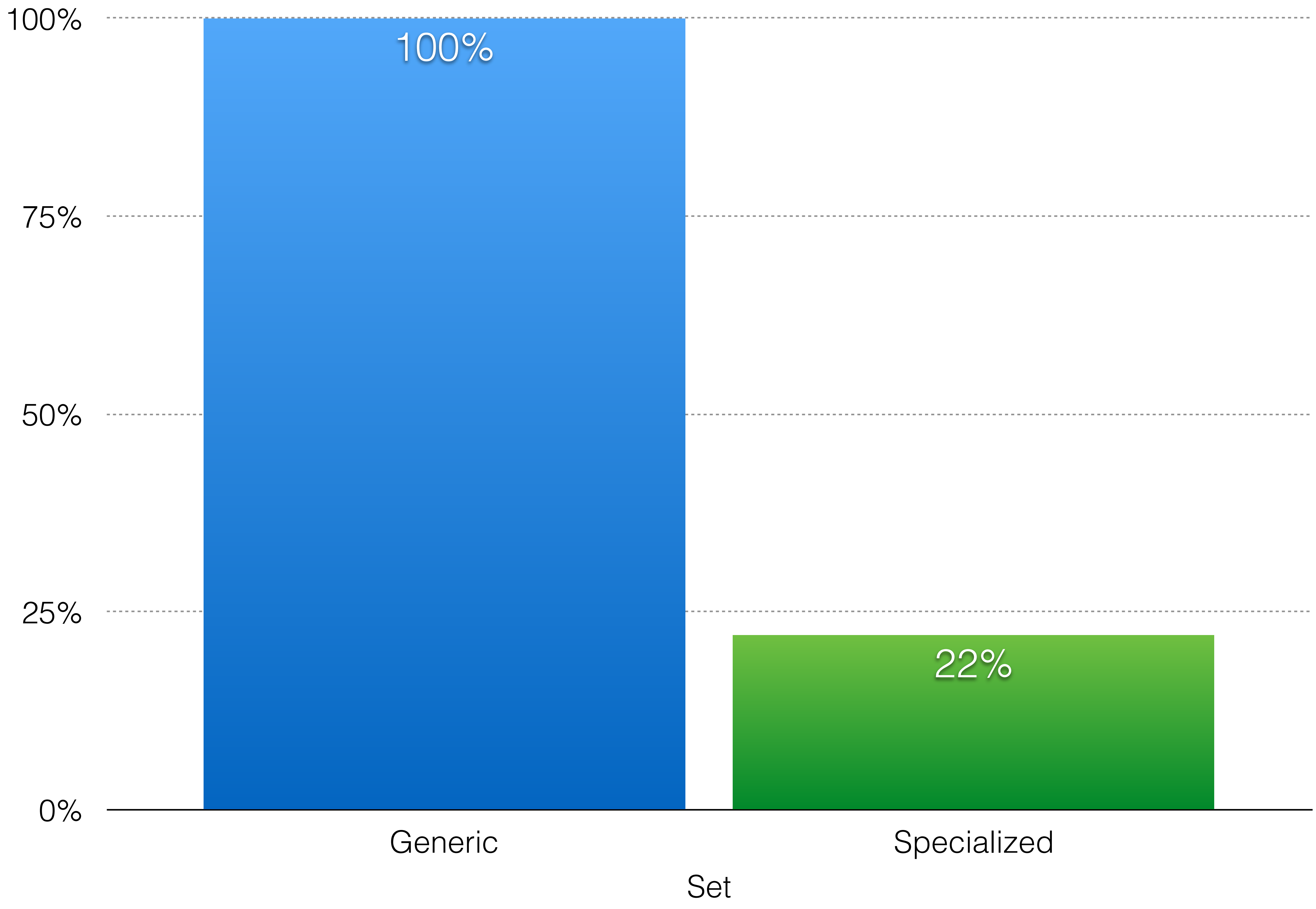
Code Specialization for Memory Efficient Hash Tries

Michael Steindorfer, Jurgen Vinju
Centrum Wiskunde & Informatica, Amsterdam, The Netherlands


100%
75%
50%
25%
0%
100%
45%
Generic
Specialized
Map


100%
75%
50%
25%
0%
100%
22%
Generic
Specialized
Set

- **memory usage** vs **runtime**
- **size** of source code or binary
- **platform specifics**

Hash Tries

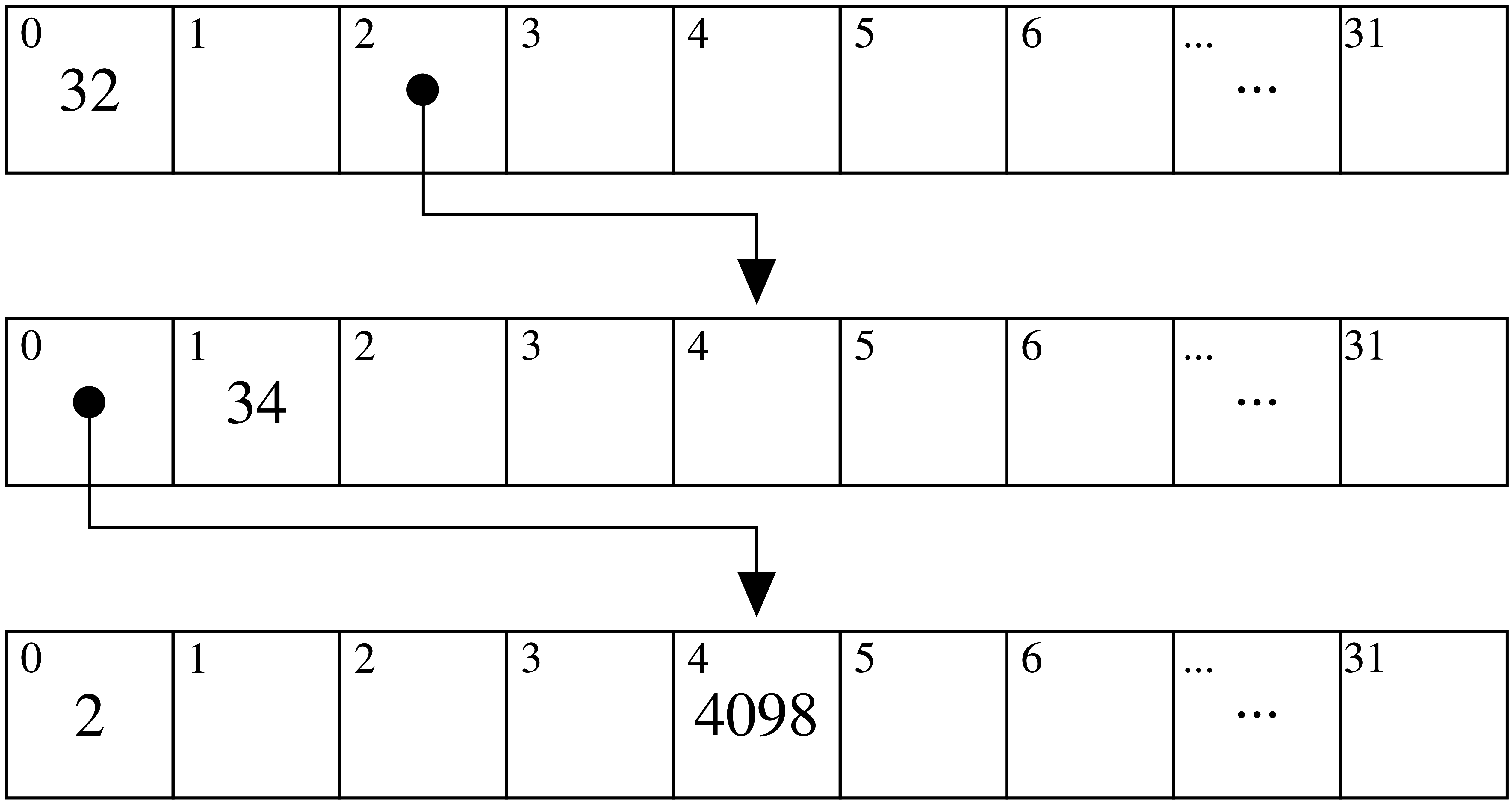
Hash Tries

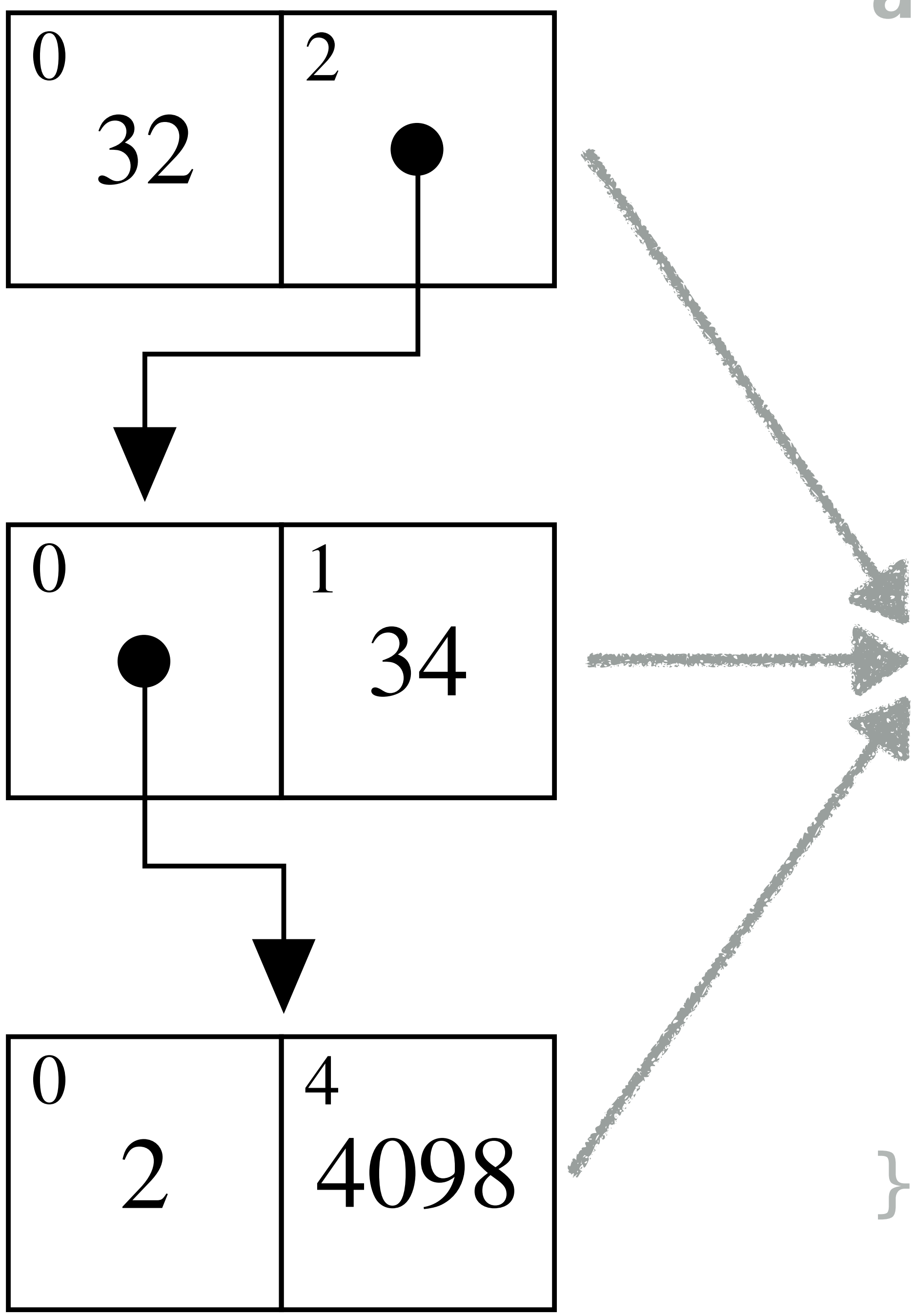
Fast Immutable Data Structures on the JVM

Hash Tries

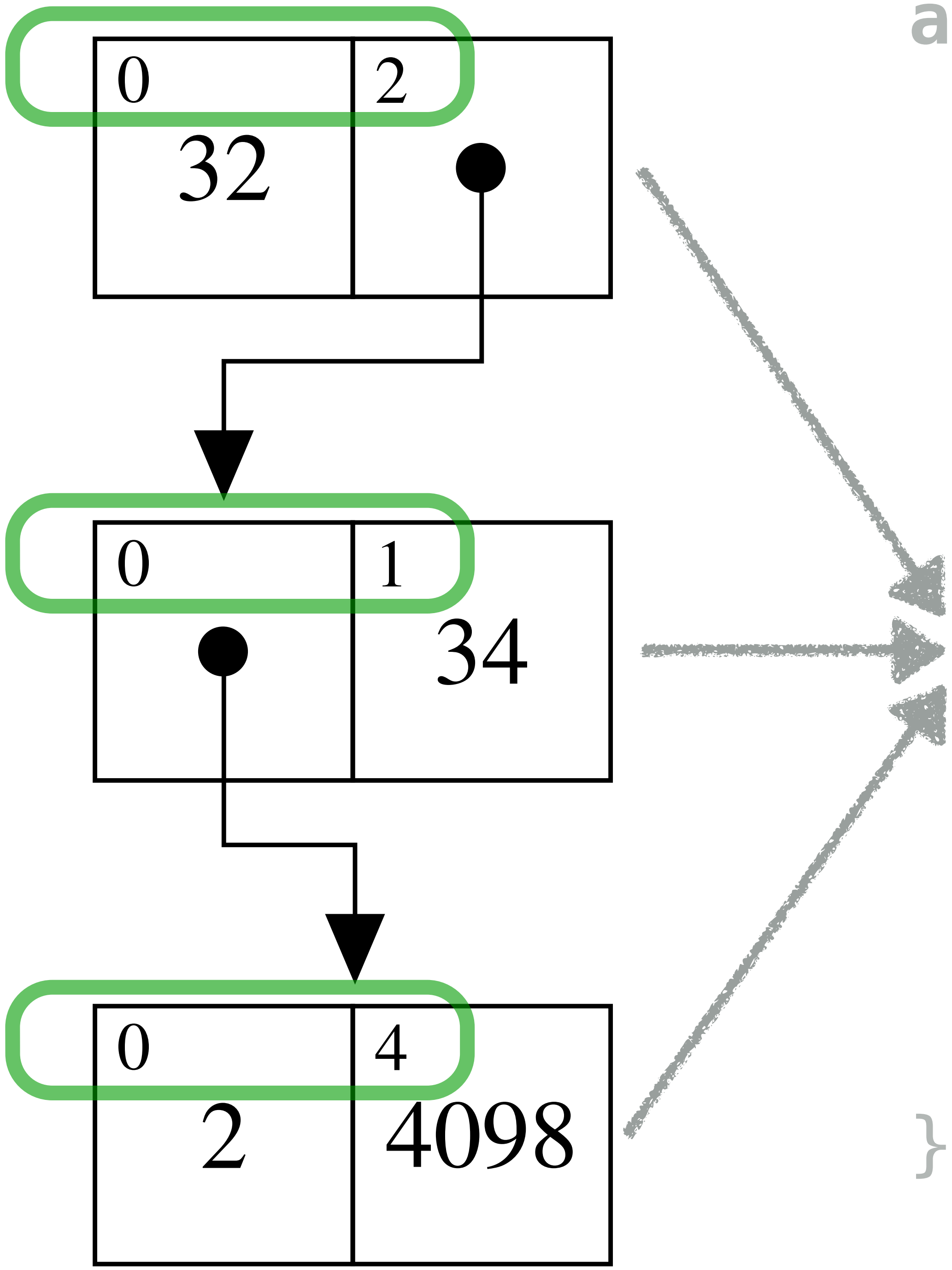
(Wide) Hash-Prefix Trees with Array Nodes

{32, 2, 4098, 34}

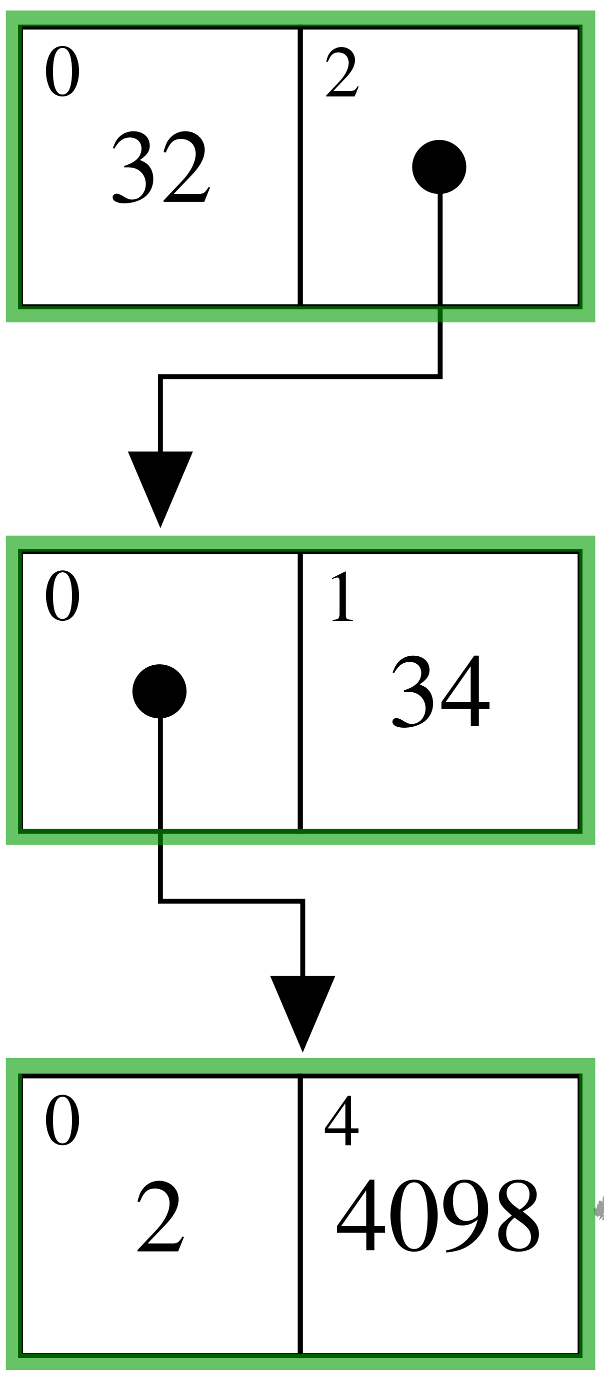

0
32
1
2
3
4
5
6
...
...
31
0
1
34
2
3
4
5
6
...
...
31
0
2
1
2
3
4
4098
5
6
...
...
31




```java
abstract class TrieSet
    implements java.util.Set {

  TrieNode root;
  int size;

  class TrieNode {
    int bitmap;
    Object[] contentAndSubTries;
  }
}
```




```java
abstract class TrieSet
    implements java.util.Set {

  TrieNode root;
  int size;

  class TrieNode {
    int bitmap;
    Object[] contentAndSubTries;
  }
}
```




```java
abstract class TrieSet
    implements java.util.Set {

  TrieNode root;
  int size;

  class TrieNode {
    int bitmap;
    Object[] contentAndSubTries;
  }
}
```

```java
class TrieNode {
  int bitmap;
  Object[] contentAndSubTries;
}
```

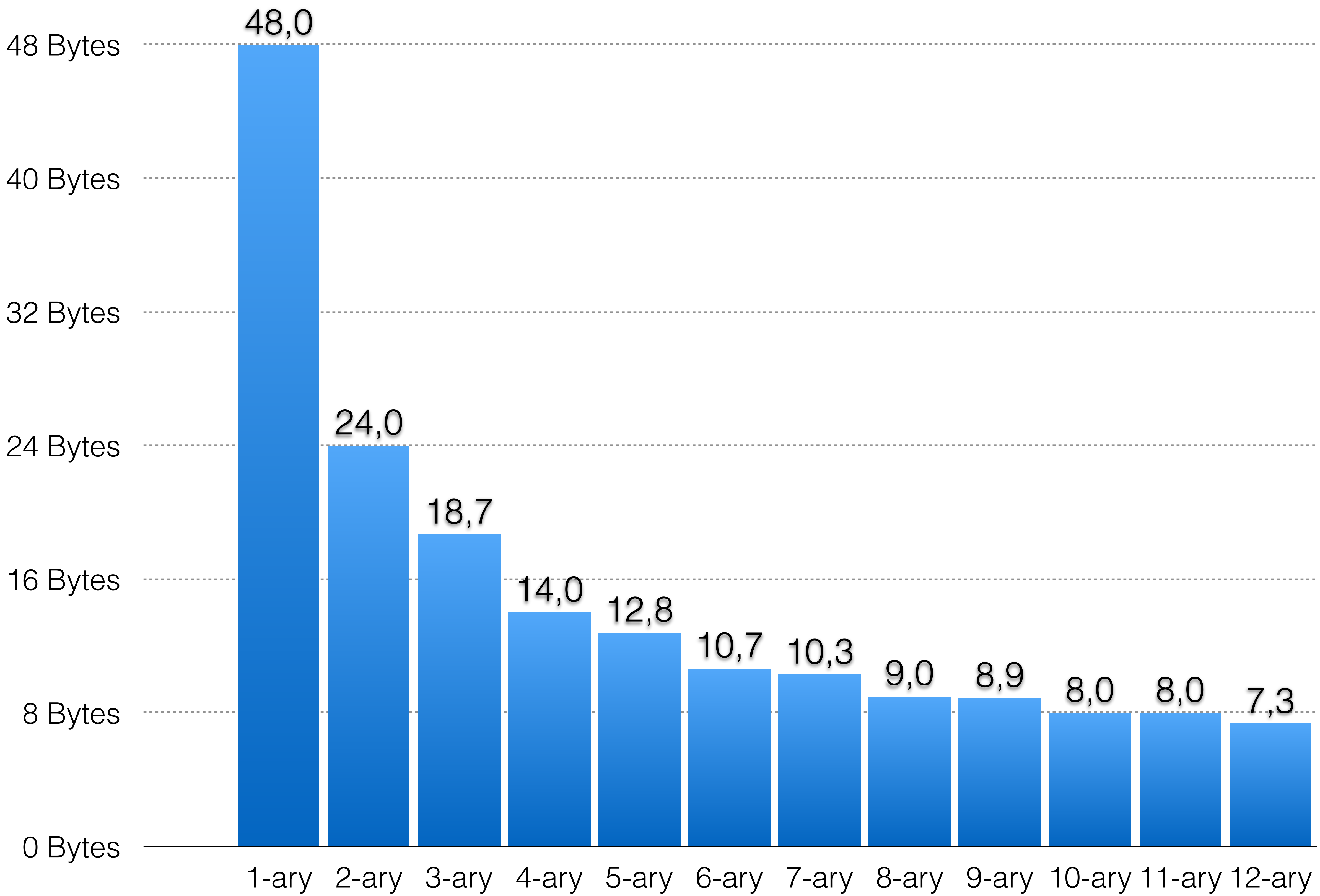
```java
class TrieNode {
  int bitmap;
  Object[] contentAndSubTries;
}
```

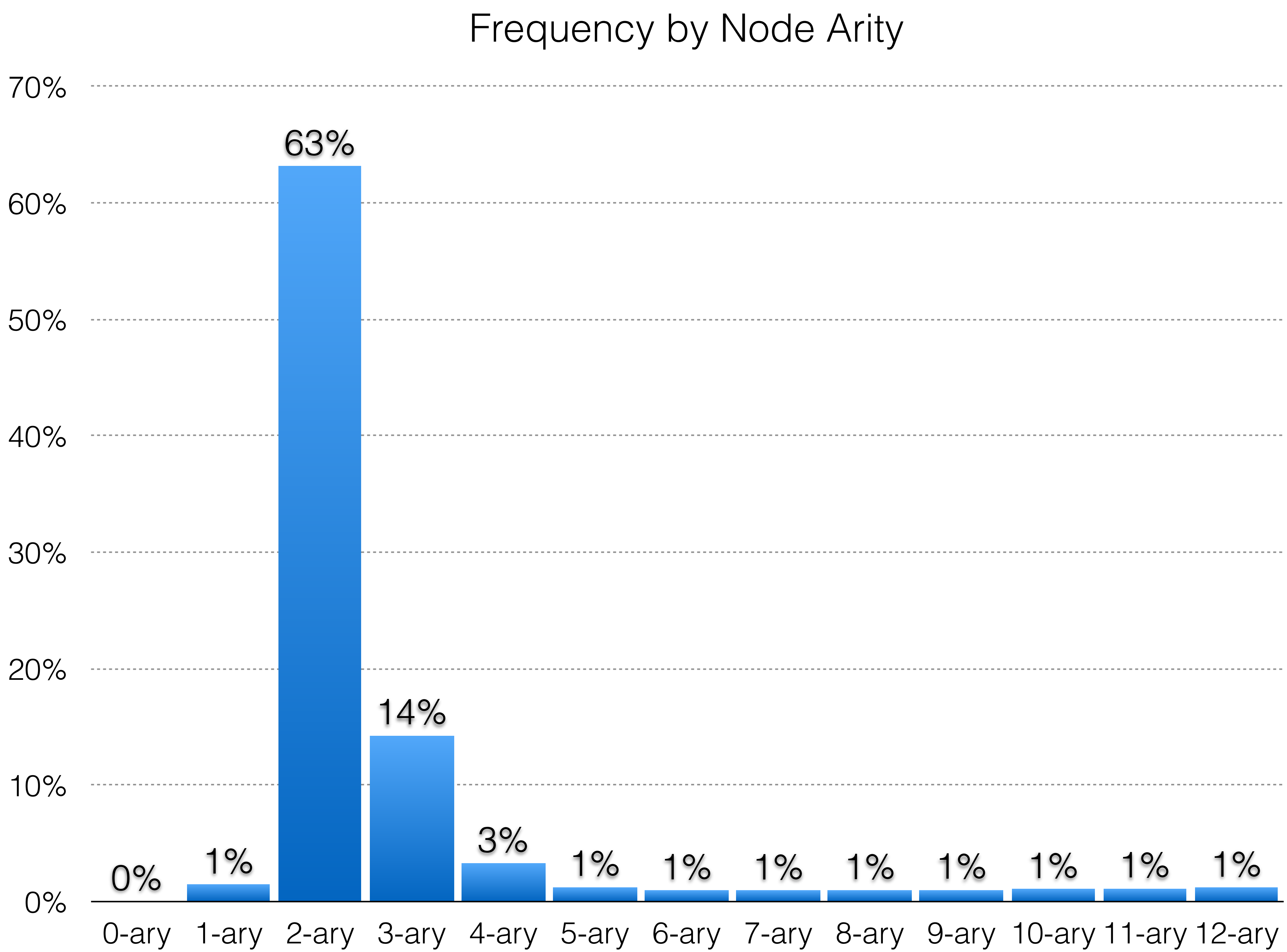
```java
...
class NodeNode extends TrieNode {
  int bitmap;
  TrieNode nodeAtIndex0;
  TrieNode nodeAtIndex1;
}
class ElementNode extends TrieNode {
  int bitmap;
  Object keyAtIndex0;
  TrieNode nodeAtIndex1;
}
class NodeElement extends TrieNode {
  int bitmap;
  TrieNode nodeAtIndex0;
  Object keyAtIndex1;
}
...
```

Exponential

Number of Specializations

Memory Overhead per Pointer (Set, 32-bit)



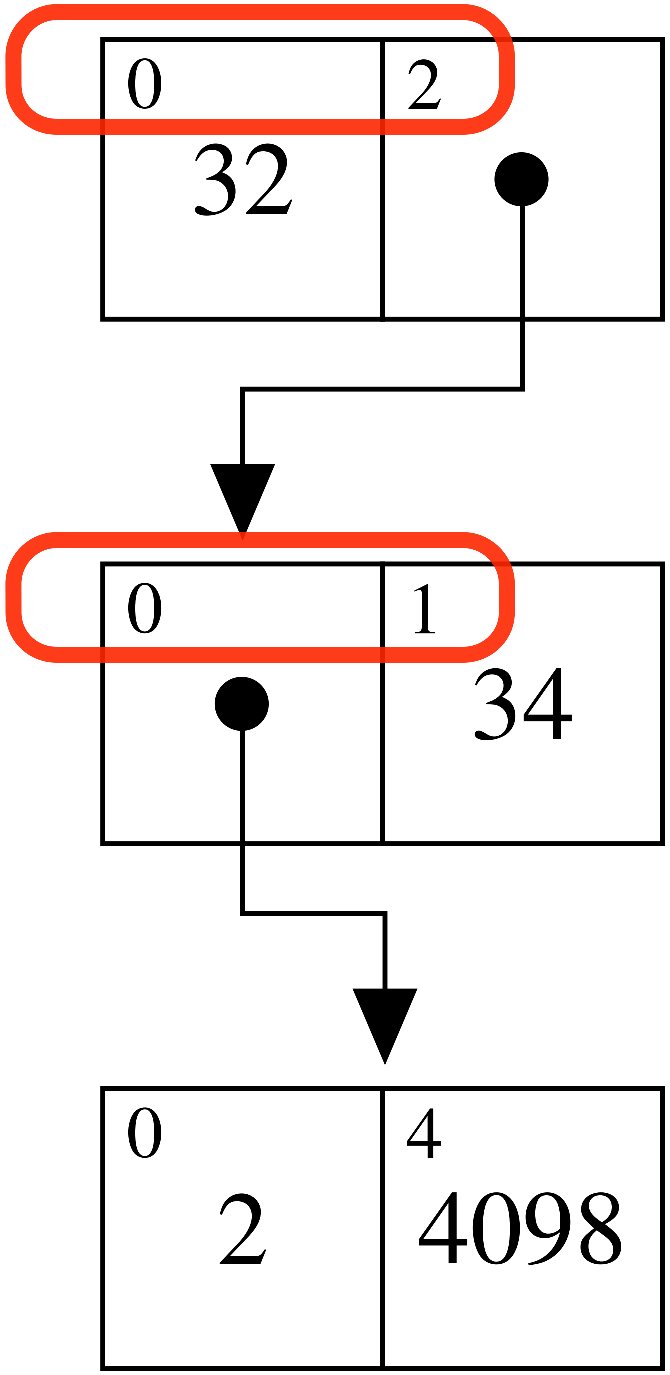

Frequency by Node Arity
70%
60%
50%
40%
30%
20%
10%
0%
0%
1%
63%
14%
3%
1%
1%
1%
1%
1%
1%
1%
1%
0-ary
1-ary
2-ary
3-ary
4-ary
5-ary
6-ary
7-ary
8-ary
9-ary
10-ary
11-ary
12-ary

Arities	% of Nodes
≤4	**82%**
≤8	**86%**
≤12	**90%**

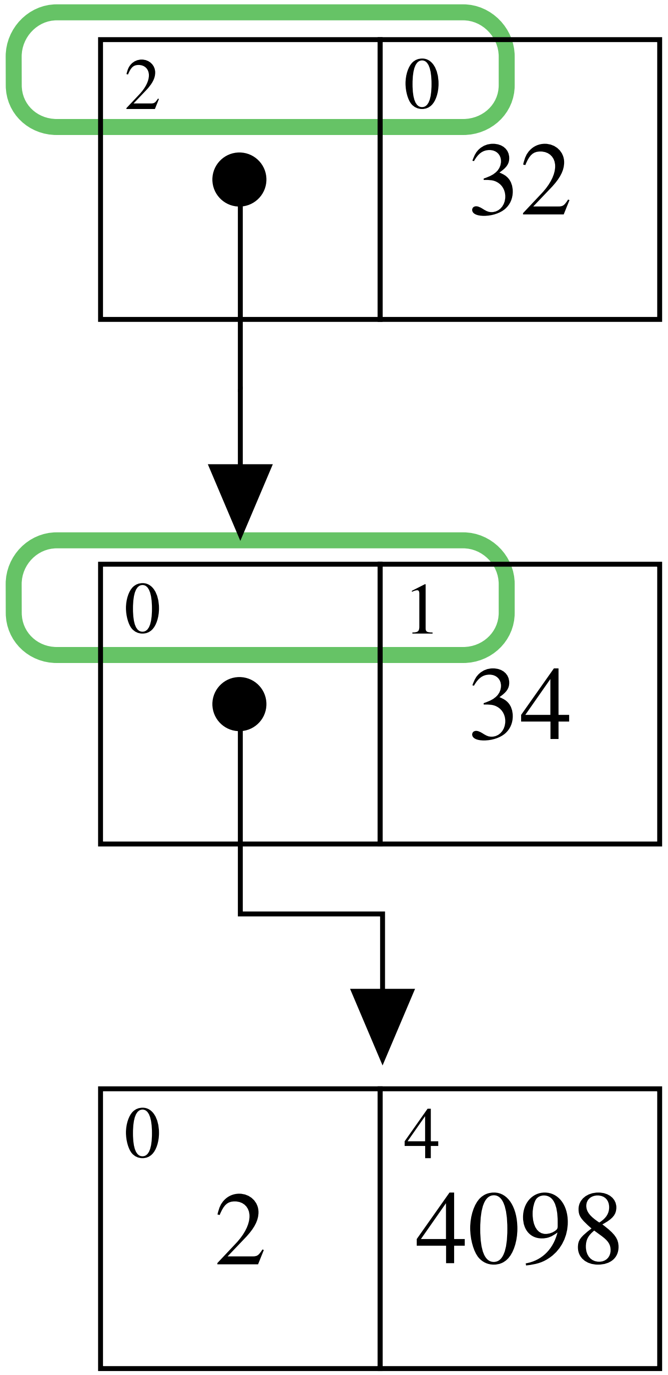
Arities	Specializations
≤4	**31**
≤8	**511**
≤12	**8191**

Avoiding
Permutations

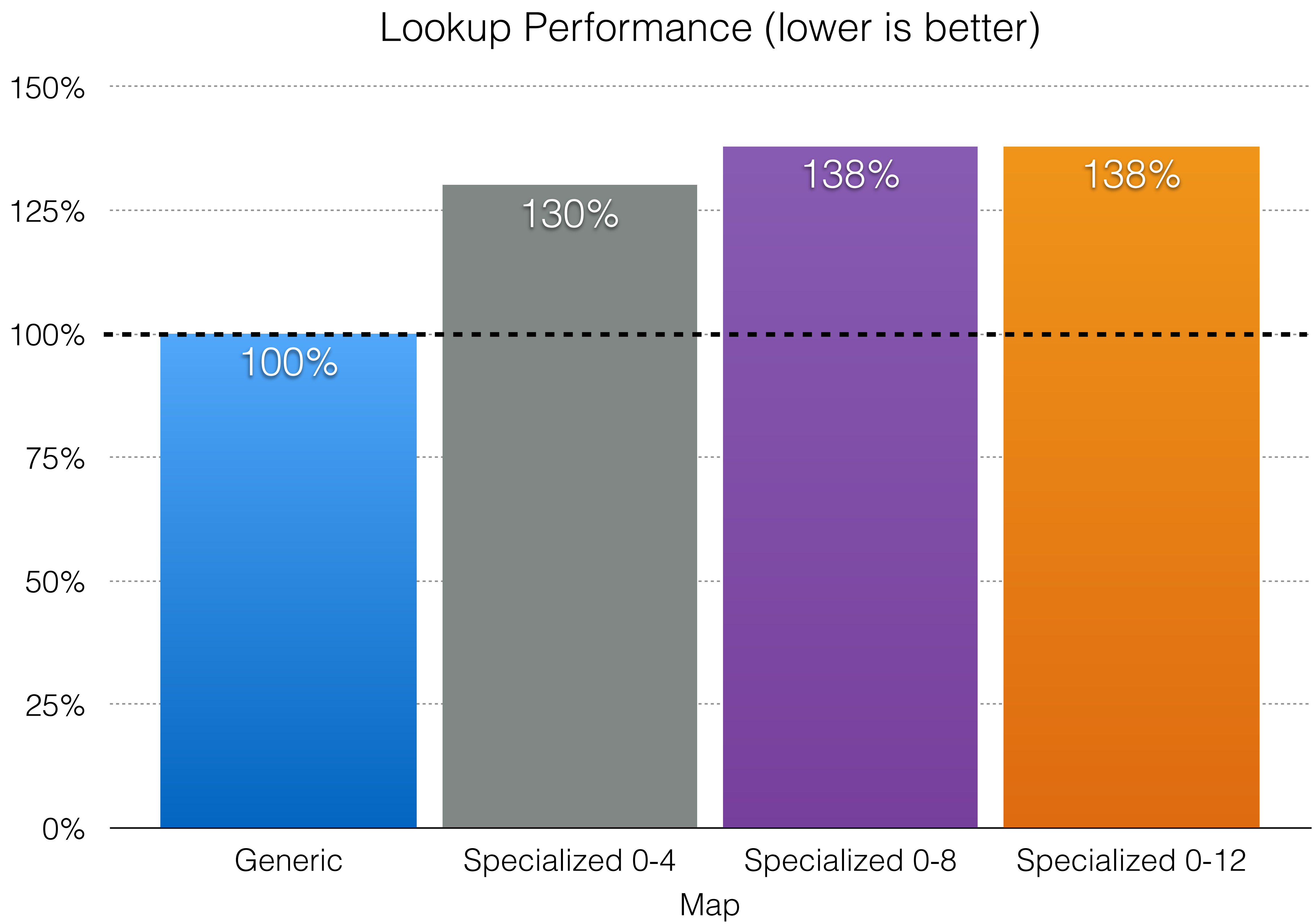
Arities	Specializations
≤4	**15** (31)
≤8	**45** (511)
≤12	**91** (8191)

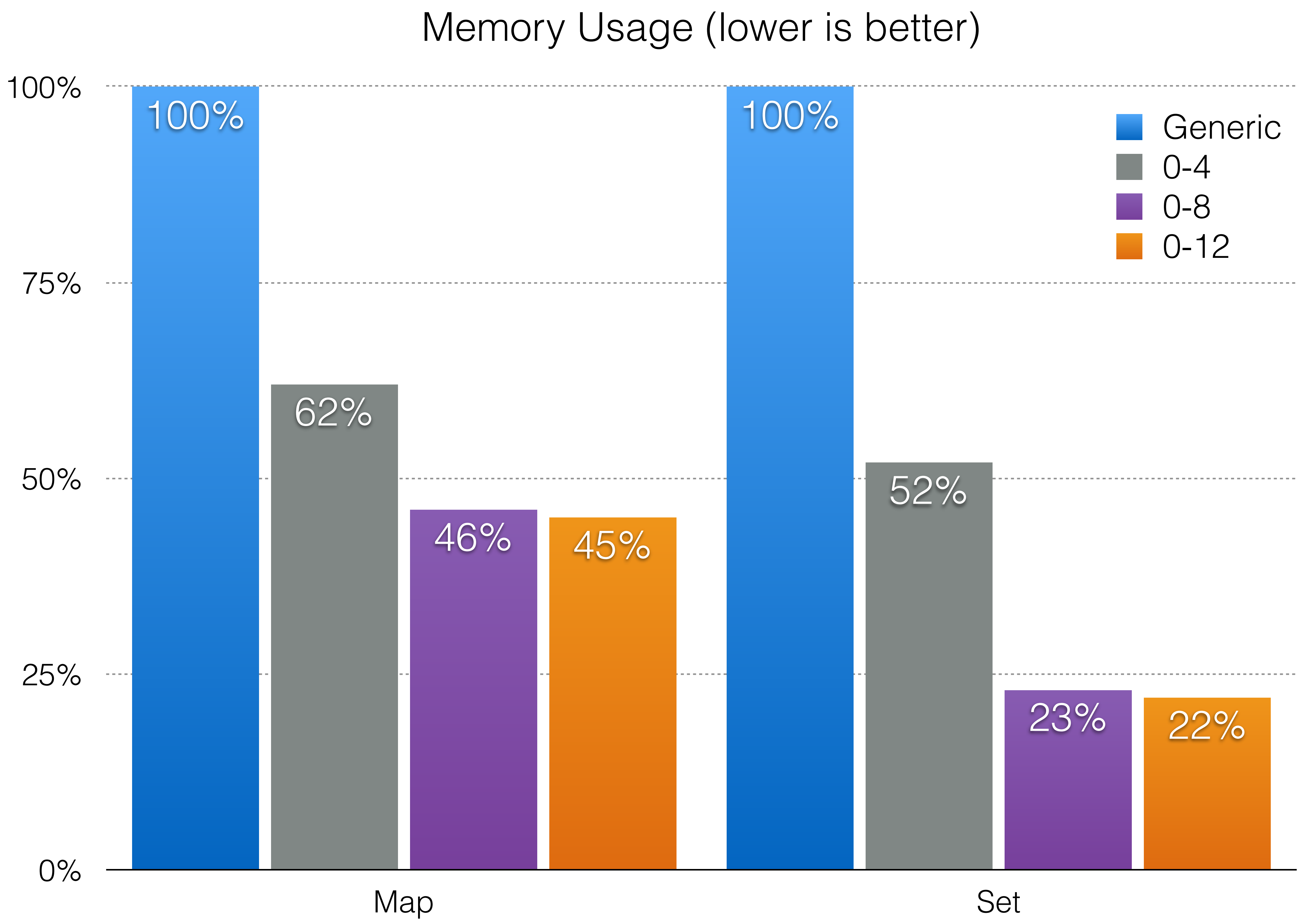


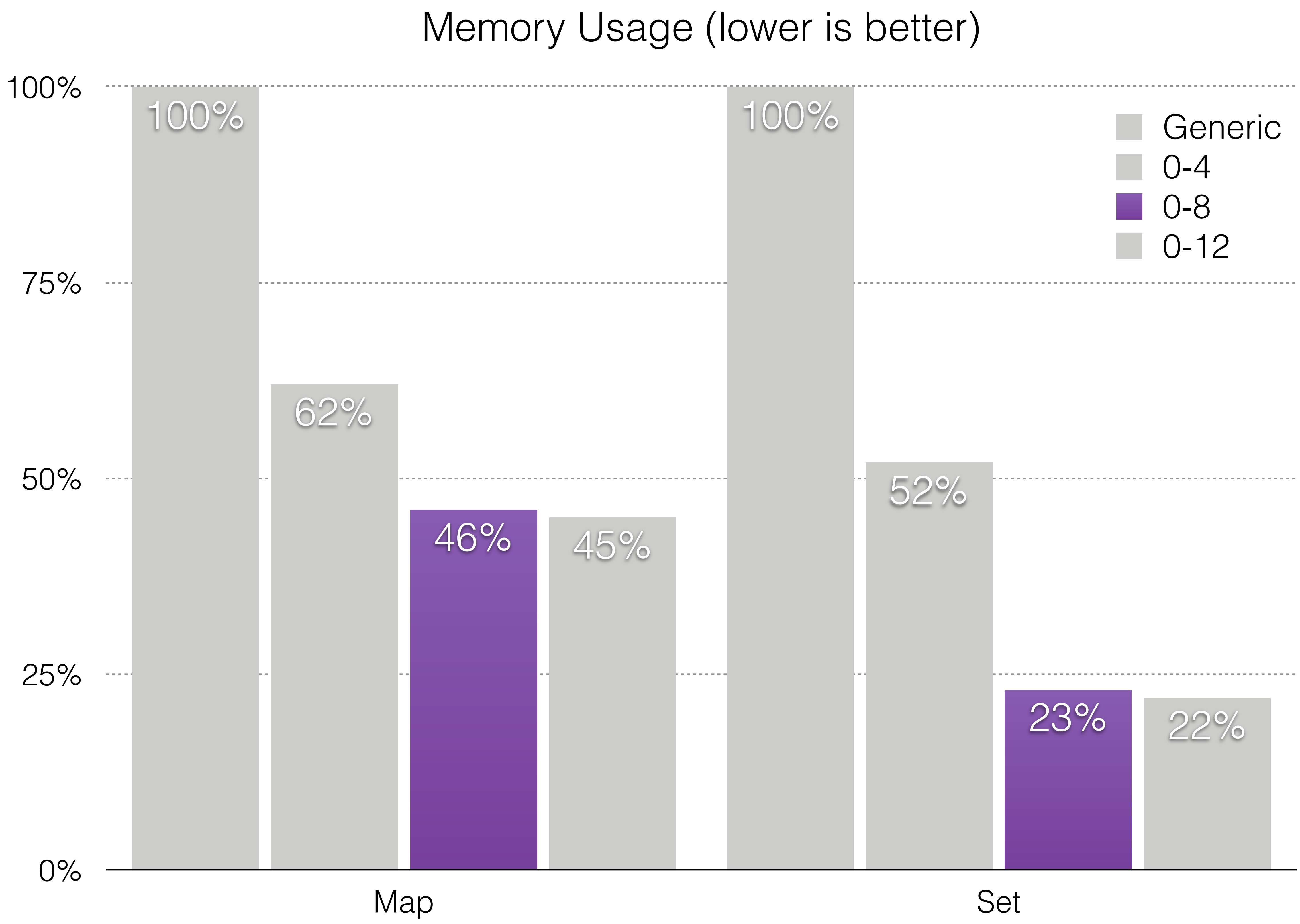

```
abstract class TrieSet implements java.util.Set {
   TrieNode root;  int size;

   interface TrieNode { ... }
   ...
   class NodeNode extends TrieNode {
      int bitmap;
      TrieNode nodeAtIndex0;
      TrieNode nodeAtIndex1;
   }
   class ElementNode extends TrieNode {
      int bitmap;
      Object keyAtIndex0;
      TrieNode nodeAtIndex1;
   }
   class NodeElement extends TrieNode {
      int bitmap;
      TrieNode nodeAtIndex0;
      Object keyAtIndex1;
   }
   class ElementElement extends TrieNode {
      int bitmap;
      Object keyAtIndex0;
      Object keyAtIndex1;
   }
   ...
}
```

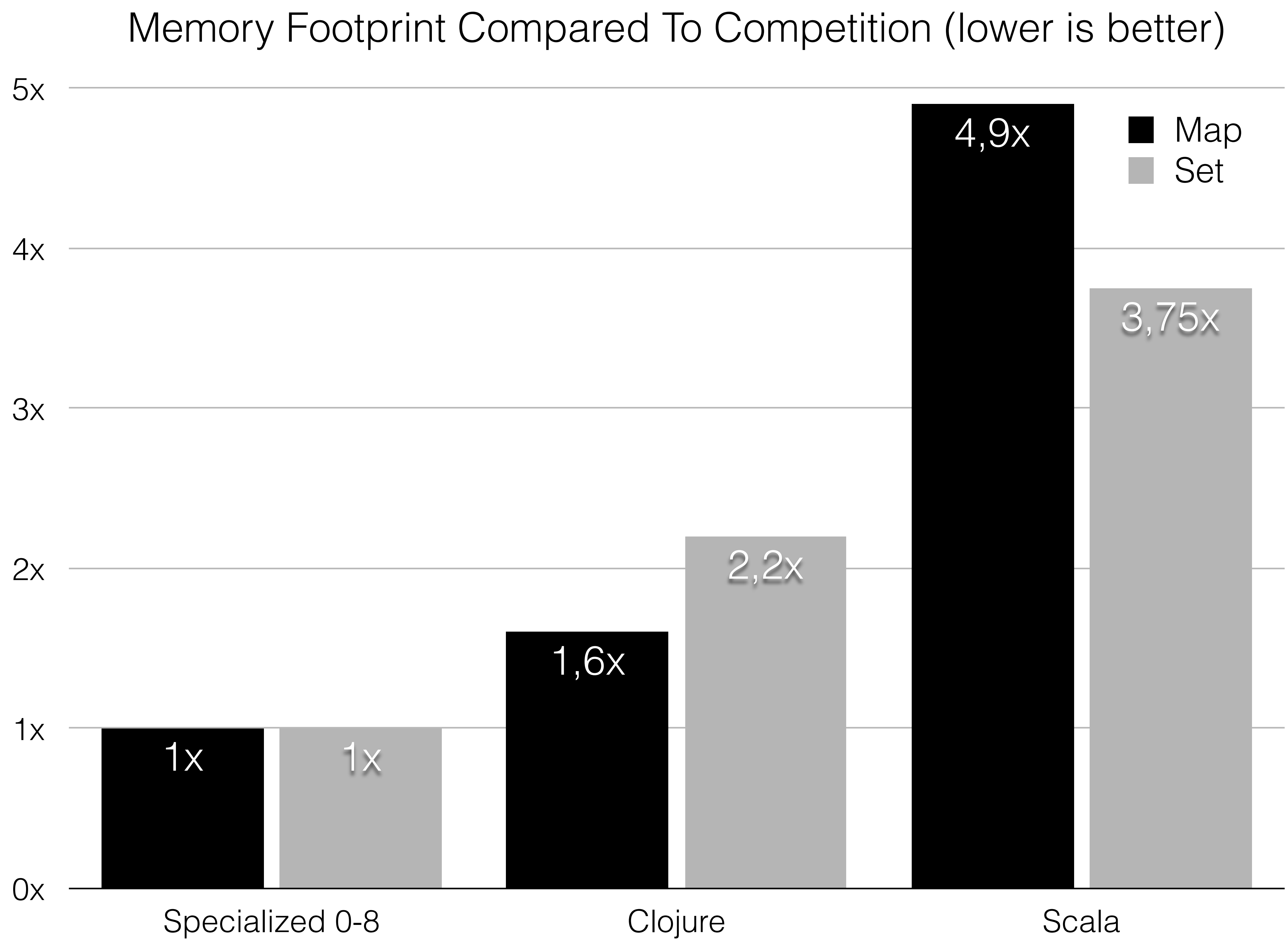



```java
abstract class TrieSet implements java.util.Set {
    TrieNode root;  int size;

    interface TrieNode { ... }
    ...
    class NodeNode extends TrieNode {

        byte pos1; TrieNode nodeAtPos1;
        byte pos2; TrieNode nodeAtPos2;
    }
    class ElementNode extends TrieNode {

        byte pos1; Object keyAtPos1;
        byte pos2; TrieNode nodeAtPos2;
    }
    class NodeElement extends TrieNode {

        byte pos1; TrieNode nodeAtPos1;
        byte pos2; Object keyAtPos2;
    }
    class ElementElement extends TrieNode {

        byte pos1; Object keyAtPos1;
        byte pos2; Object keyAtPos2;
    }
    ...
}
```


Lookup Performance (lower is better)
150%
125%
100%
75%
50%
25%
0%
100%
130%
138%
138%
Generic
Specialized 0-4
Specialized 0-8
Specialized 0-12
Map


Memory Usage (lower is better)
100%
75%
50%
25%
0%
100%
62%
46%
45%
100%
52%
23%
22%
Generic
0-4
0-8
0-12
Map
Set


Memory Usage (lower is better)
100%
75%
50%
25%
0%
100%
62%
46%
45%
100%
52%
23%
22%
Generic
0-4
0-8
0-12
Map
Set


Memory Footprint Compared To Competition (lower is better)
5x
4x
3x
2x
1x
0x
Map
Set
1x
1x
1,6x
2,2x
4,9x
3,75x
Specialized 0-8
Clojure
Scala

worst **hash distribution**

->

good **memory performance**

best **hash distribution**

->

worst **memory performance**

best **hash distribution**

->

best

~~worst~~ **memory performance**