A Maiden Voyage into Open Source, Datatypes, and Scala

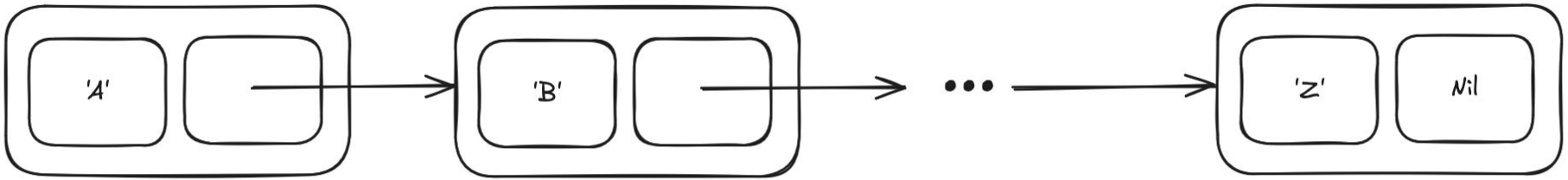
L Denney

Inception of the project

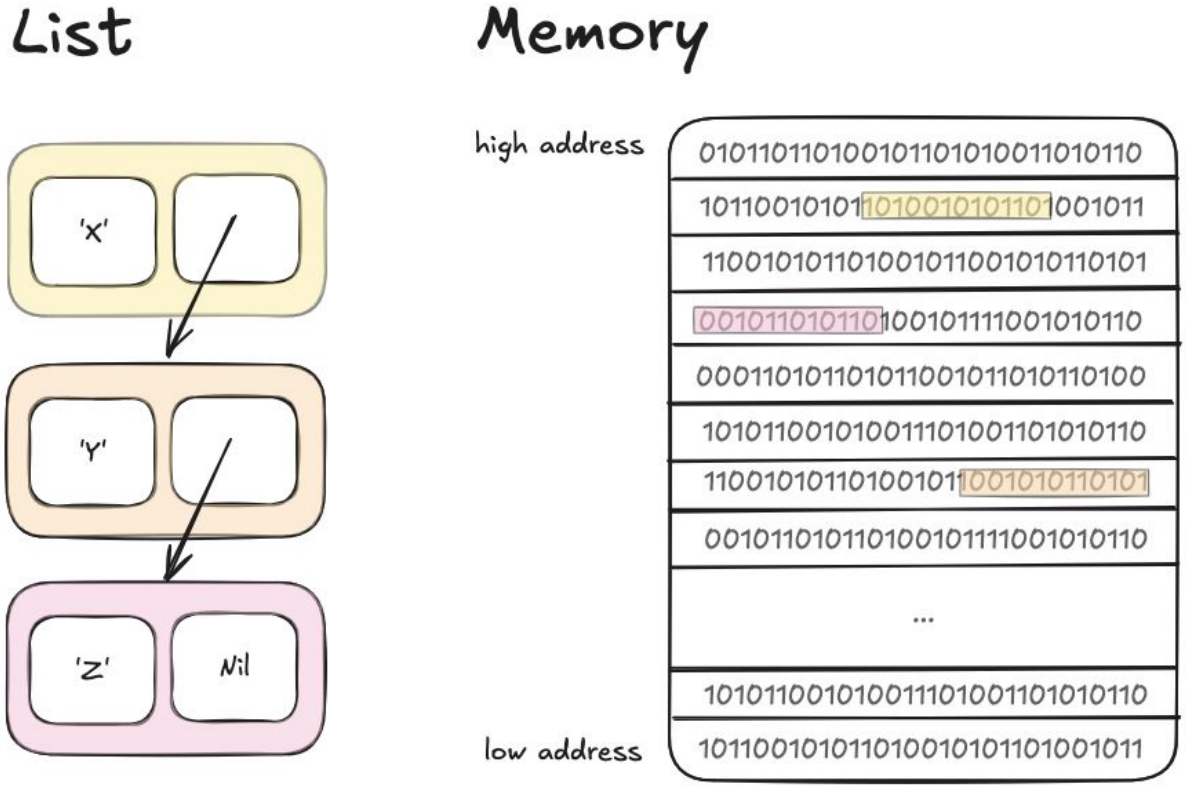
- Finished my undergrad in May 2026
- Functional Programming <3
- Interested in Open Source development
- GSoC funding



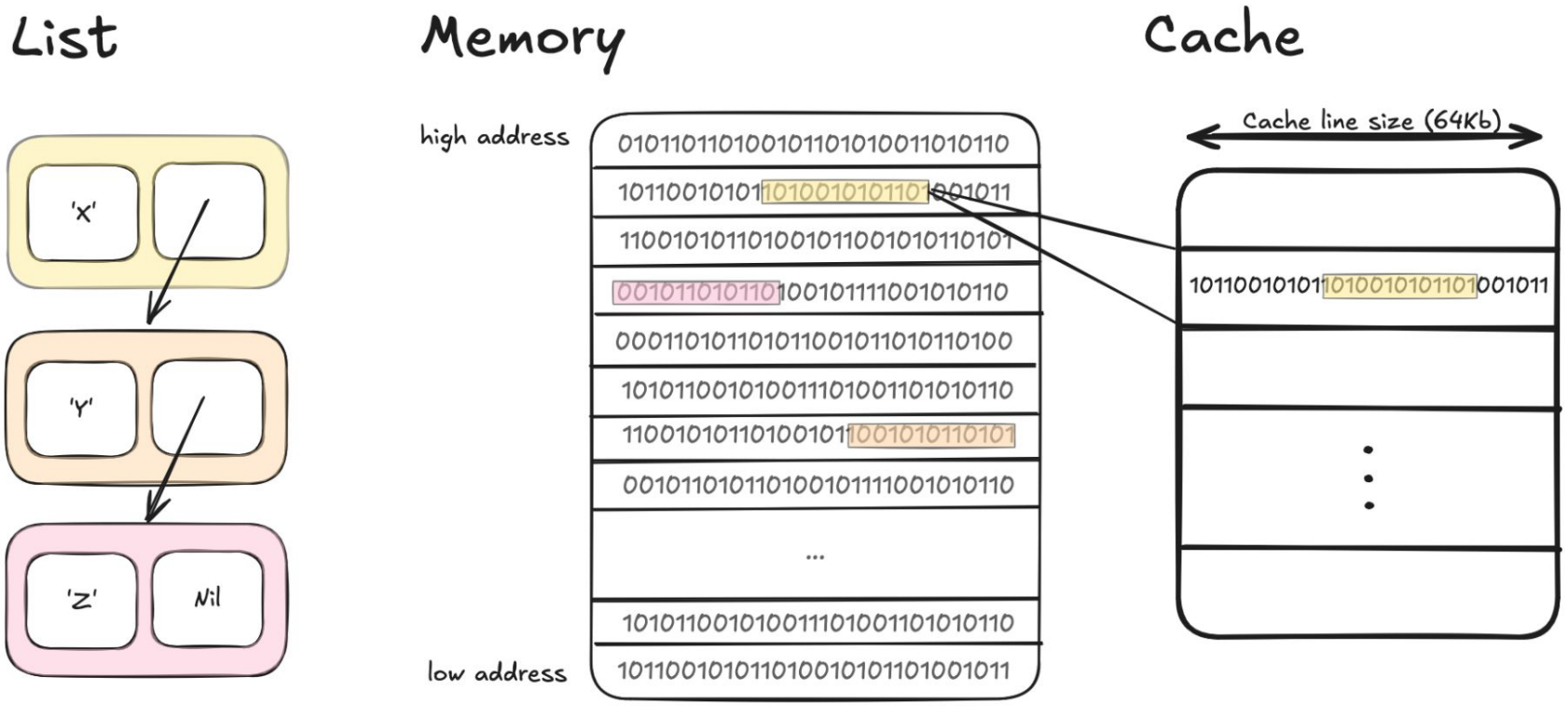
Immutable Linked List



How a linked list is stored in memory

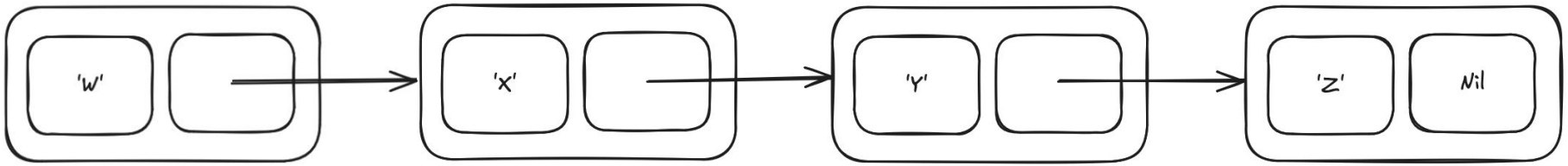


What happens when we want to interact with the list



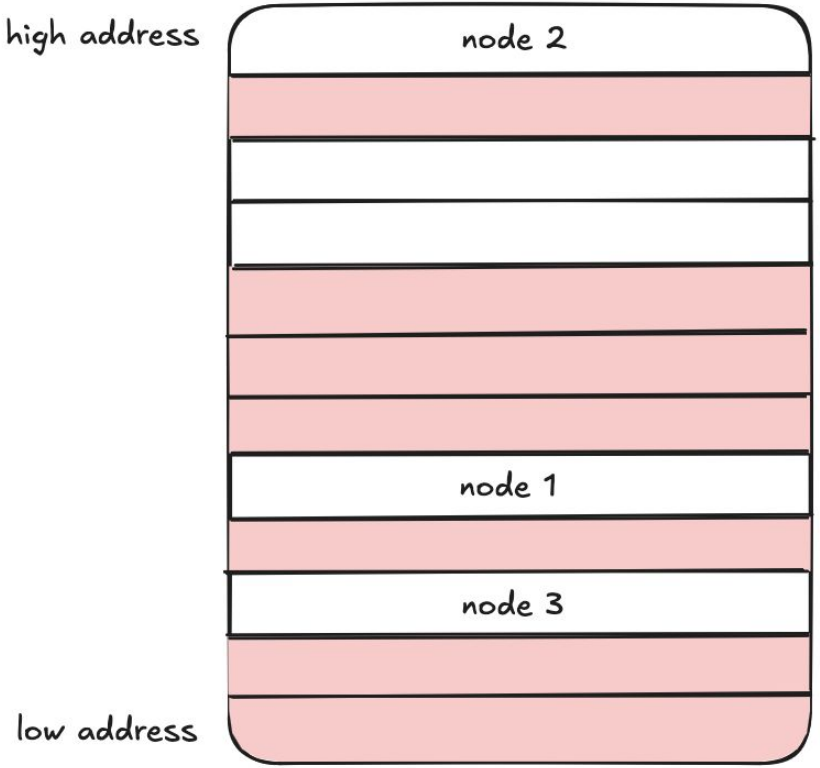
Strengths of Linked List

1. Prepending is fast



Strengths of Linked List

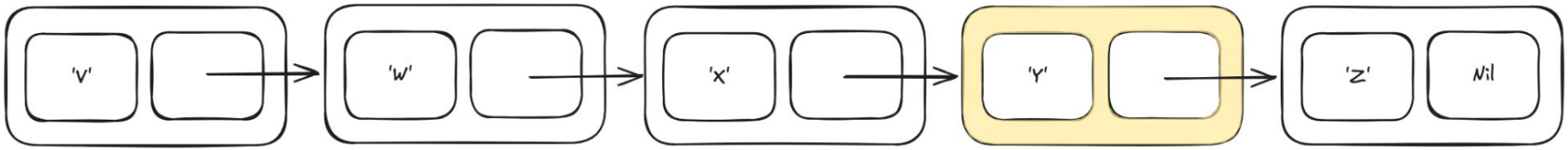
2. Easy to grow or shrink by allocating or deallocating nodes in memory



Imagine you wanted to prepend 3 new elements that each require (when wrapped up in nodes) 1 “unit” of space in memory

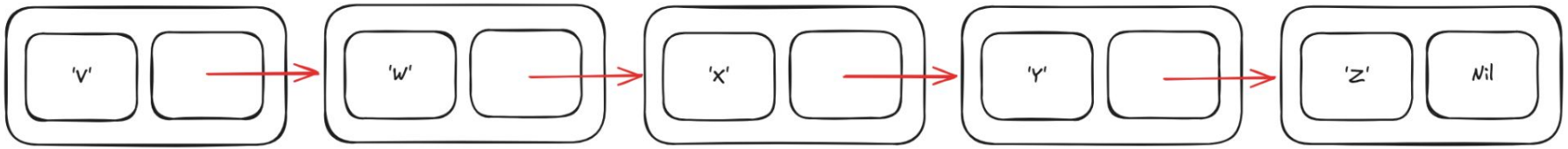
Weaknesses of Linked List

1. Random access (reading the element at some arbitrary index)



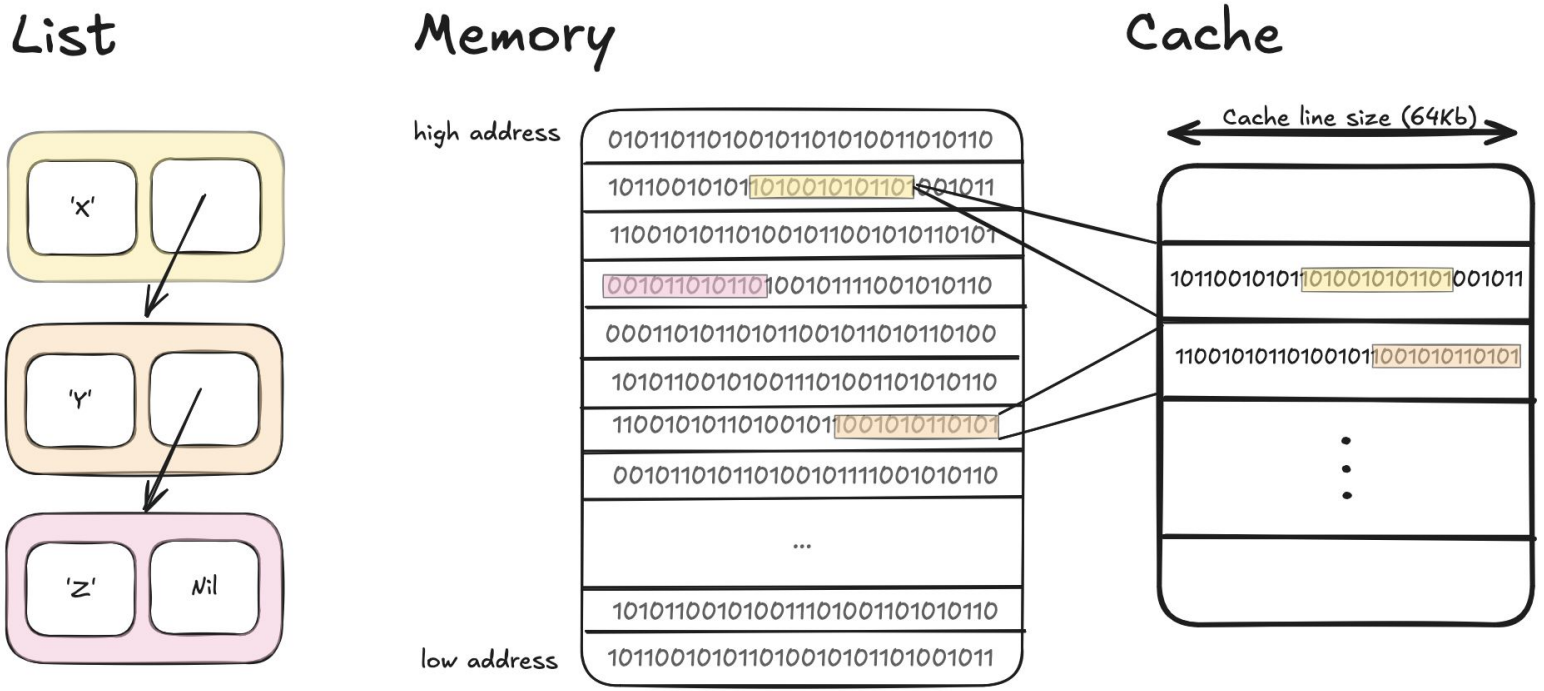
Weaknesses of Linked List

2. Iterating over the full list - ex. map, fold, filter



Weaknesses of Linked List

3. Cache inefficiency

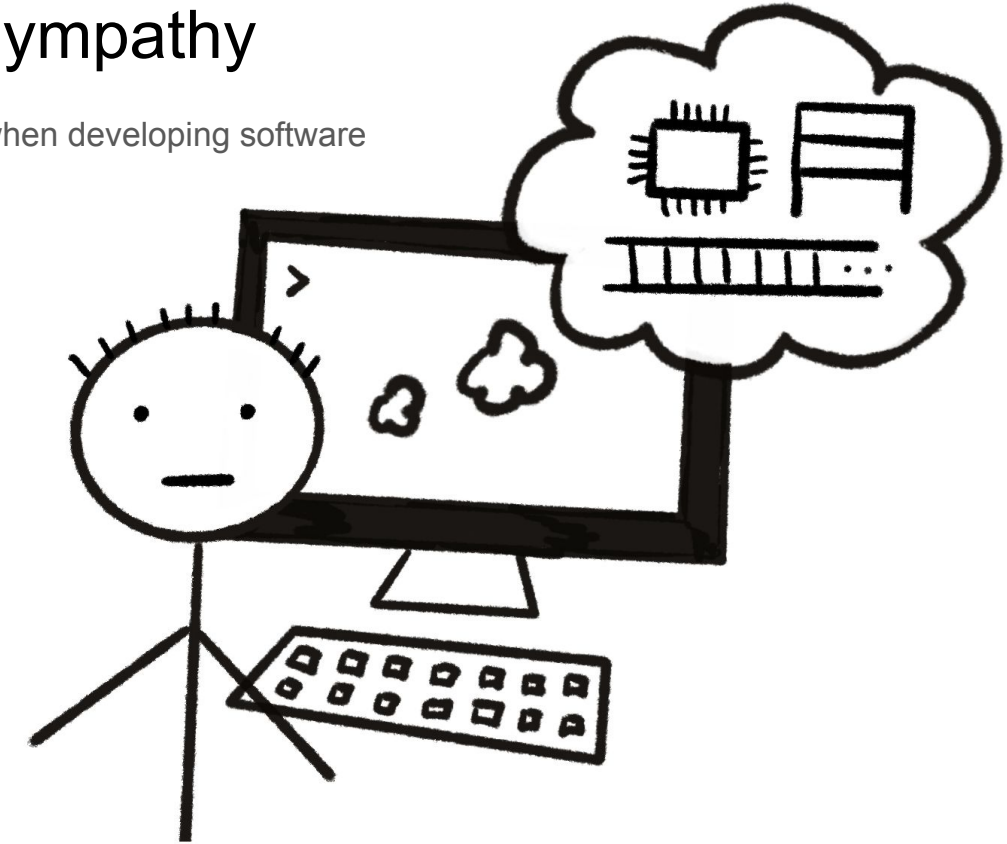


Where an immutable linked list is useful

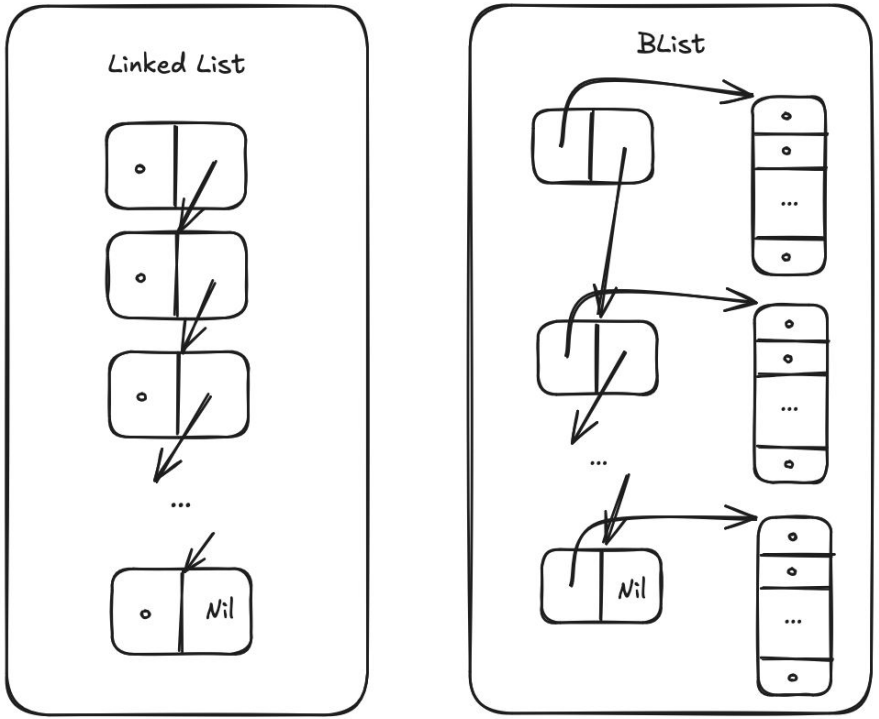
- Simulating a stack (efficient push/pop)
- Manipulating data in concurrent situations
- Within other (often concurrent) data structures
  - ex. fs2’s Channel

Mechanical Sympathy

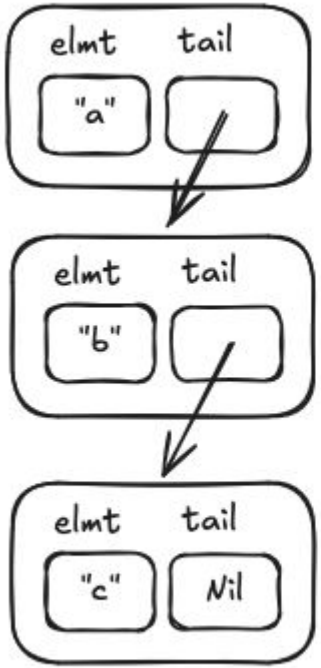
Catering to the hardware when developing software

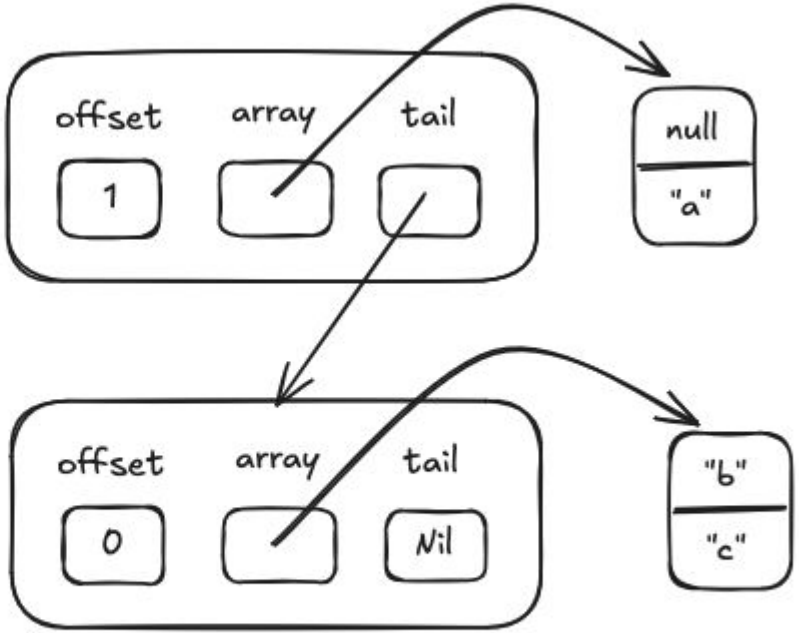


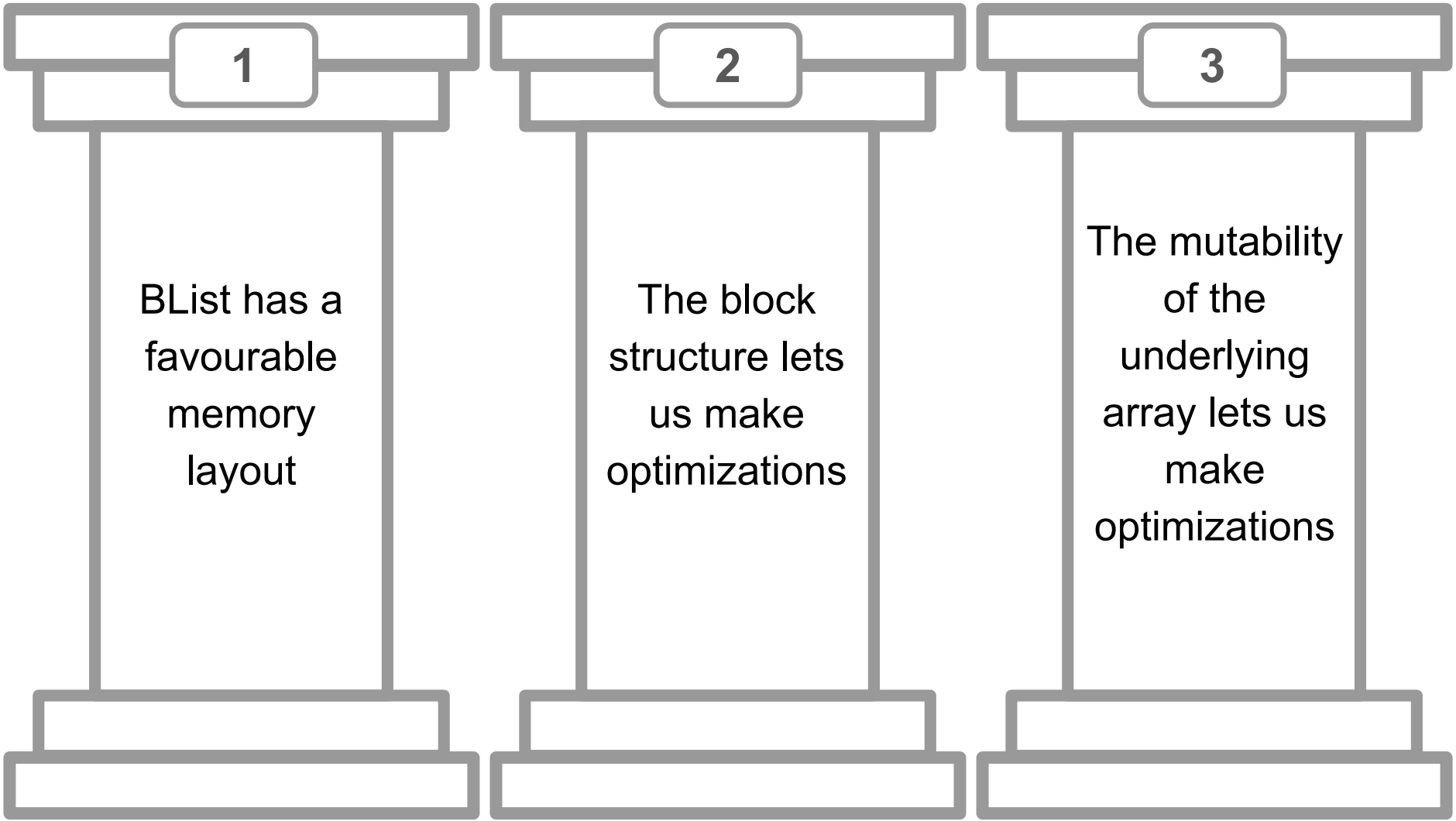
Thinking about Mechanical Sympathy… we get BList!



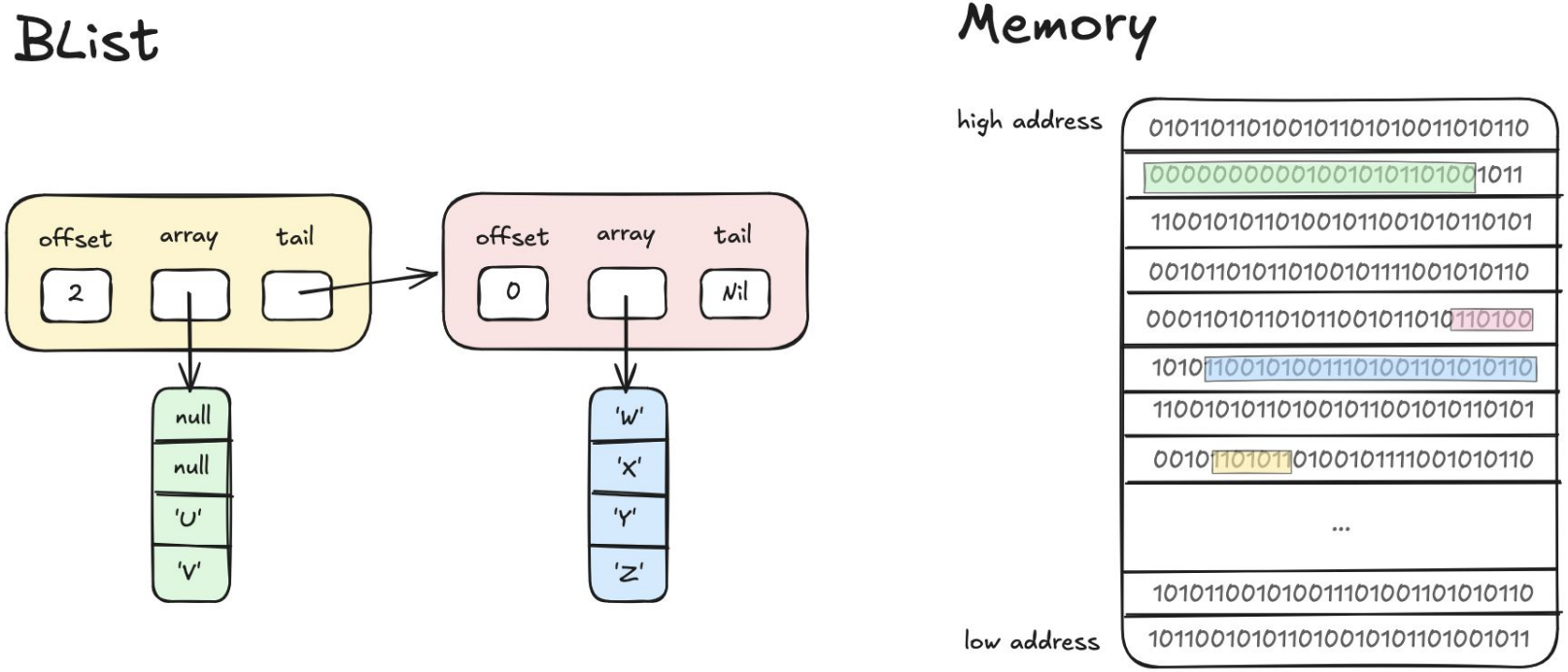
List vs. BList example






1
BList has a favourable memory layout
2
The block structure lets us make optimizations
3
The mutability of the underlying array lets us make optimizations

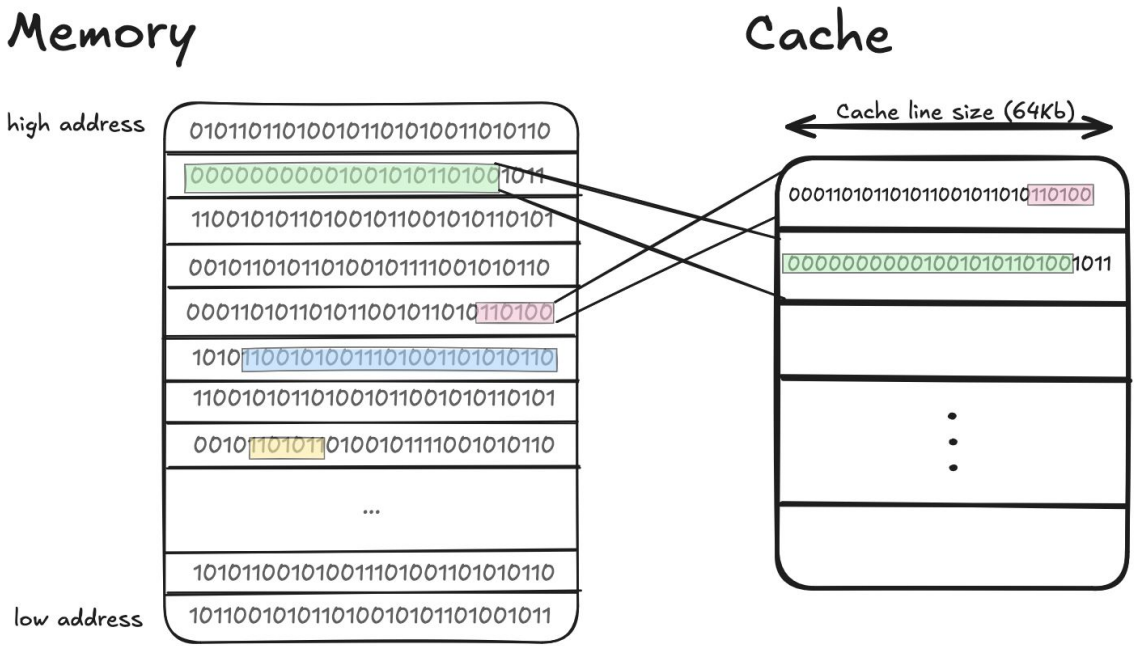
BList in Memory



“1. BList has a favourable memory layout”

BList in Cache

- Once the first element of a block is read, we can expect the others to be in the cache already
- What if fit the blocksize to fill a cache line?

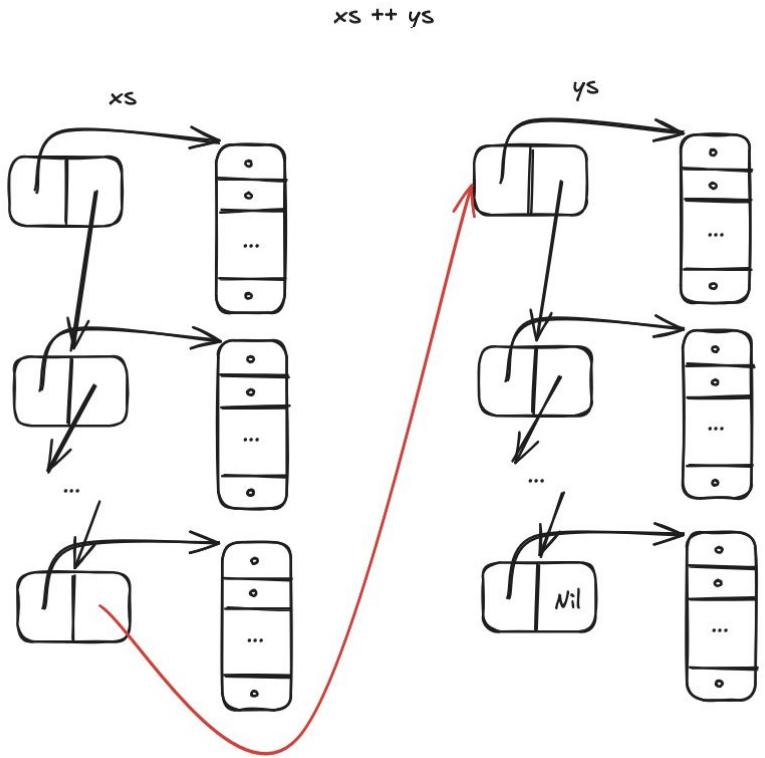


“1. BList has a favourable memory layout”

Optimizations in methods

A few classes of optimizations that BList makes possible

1. **faster skipping to a specified index in the list - splitAt, concat (++), get**
2. iteration over all elements in the list with fewer pointer dereferences - map, filter, fold, toList, toString
3. Building a new BList using appends where we can batch allocations - Builder

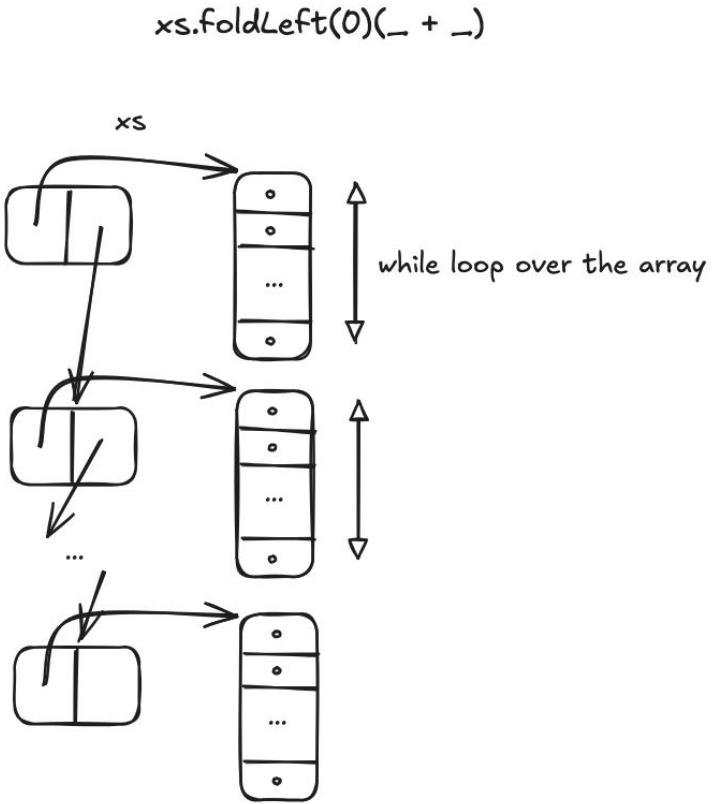


2. The block structure lets us make optimizations”

Optimizations in methods

A few classes of optimizations that BList makes possible

1. faster skipping to a specified index in the list - splitAt, concat (++), get
2. **iteration over all elements in the list with fewer pointer dereferences - map, filter, fold, toList, toString**
3. Building a new BList using appends where we can batch allocations - Builder

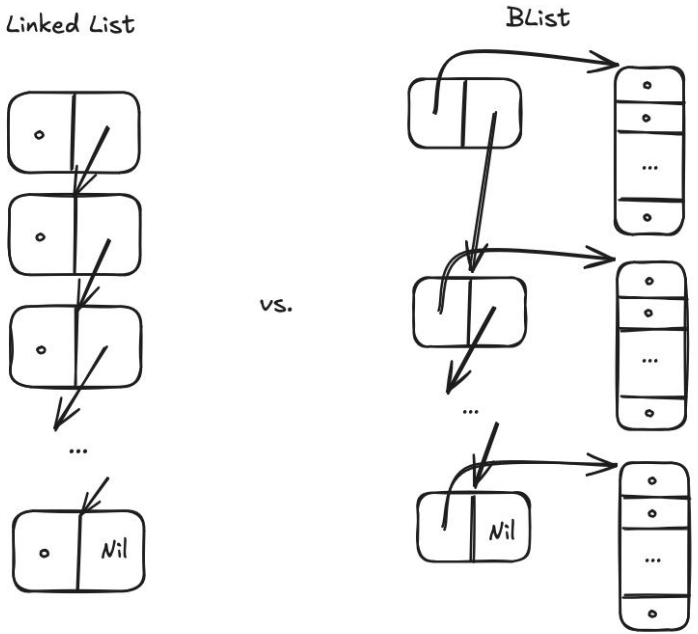


2. The block structure lets us make optimizations”

Optimizations in methods

A few classes of optimizations that BList makes possible

1. faster skipping to a specified index in the list - splitAt, concat (++), get
2. iteration over all elements in the list with fewer pointer dereferences - map, filter, fold, toList, toString
3. **Building a new BList using appends where we can batch allocations - Builder**

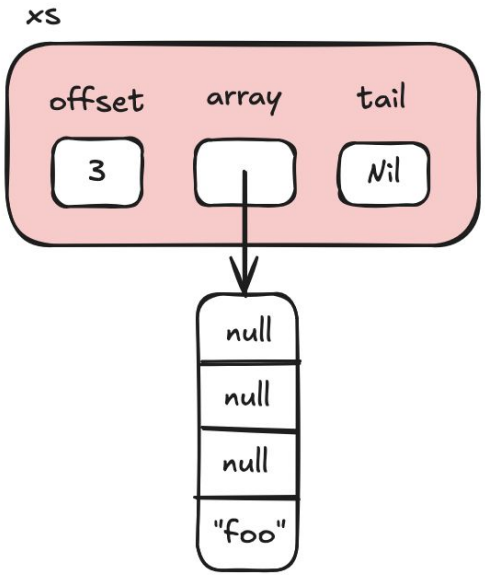


2. The block structure lets us make optimizations”

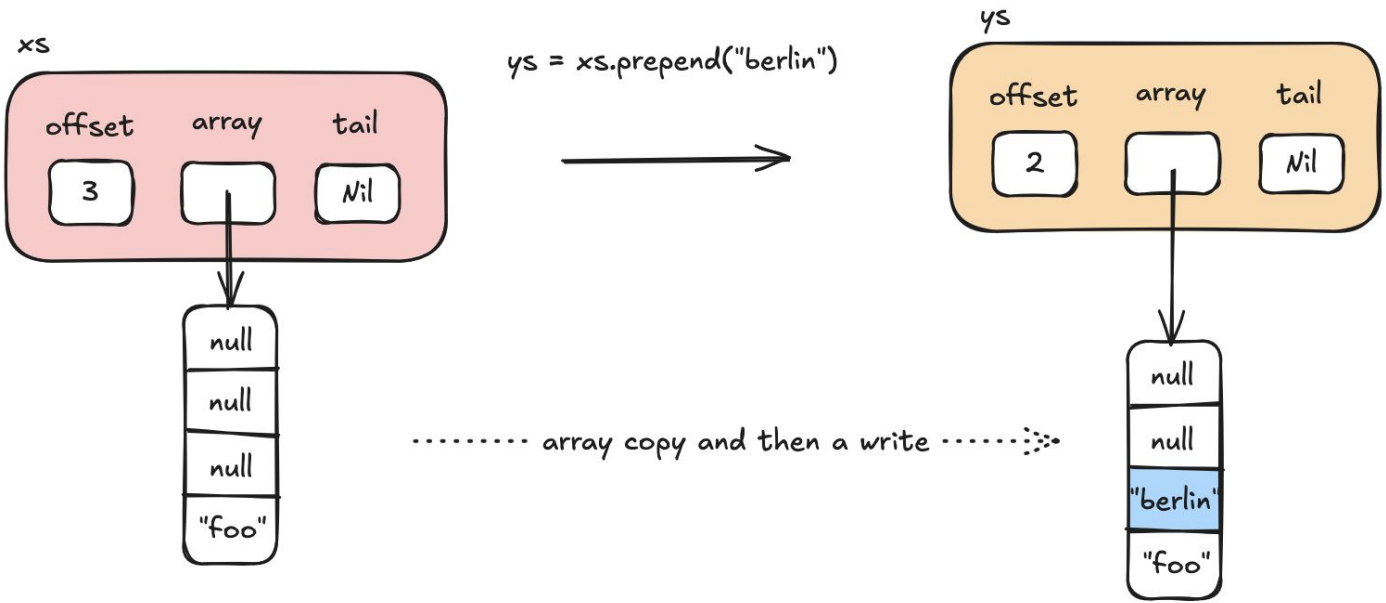
prepend

share and fork

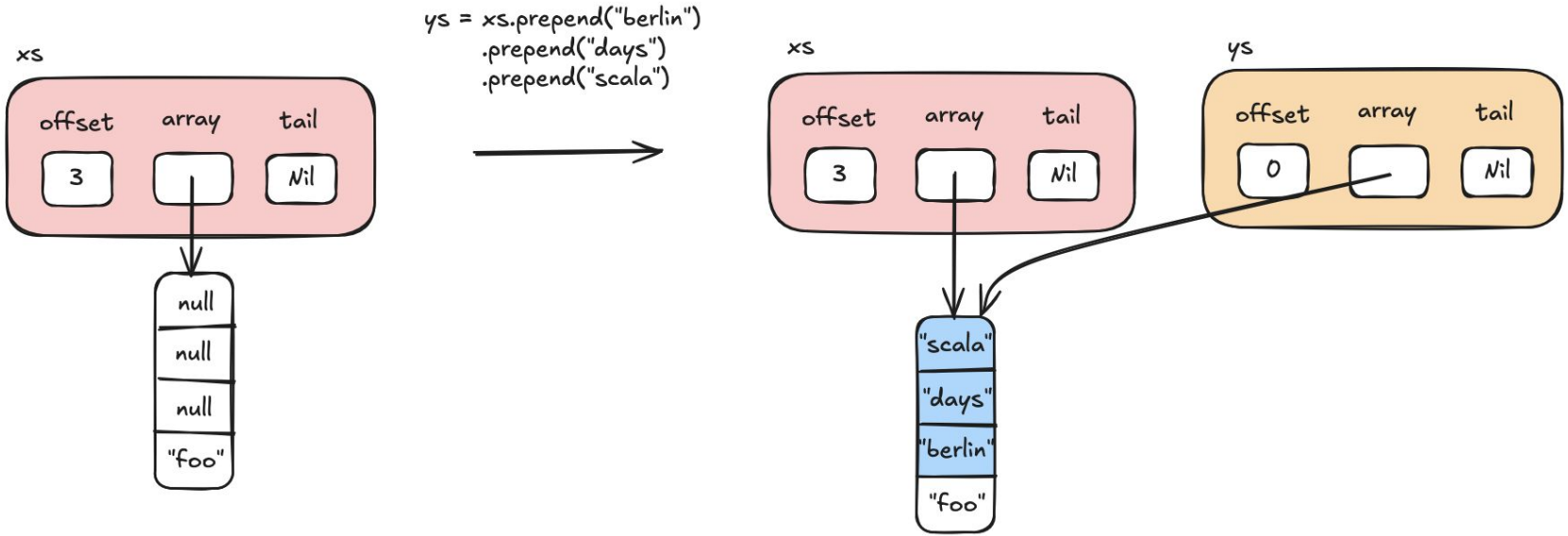
3. The mutability of the underlying array lets us make optimizations



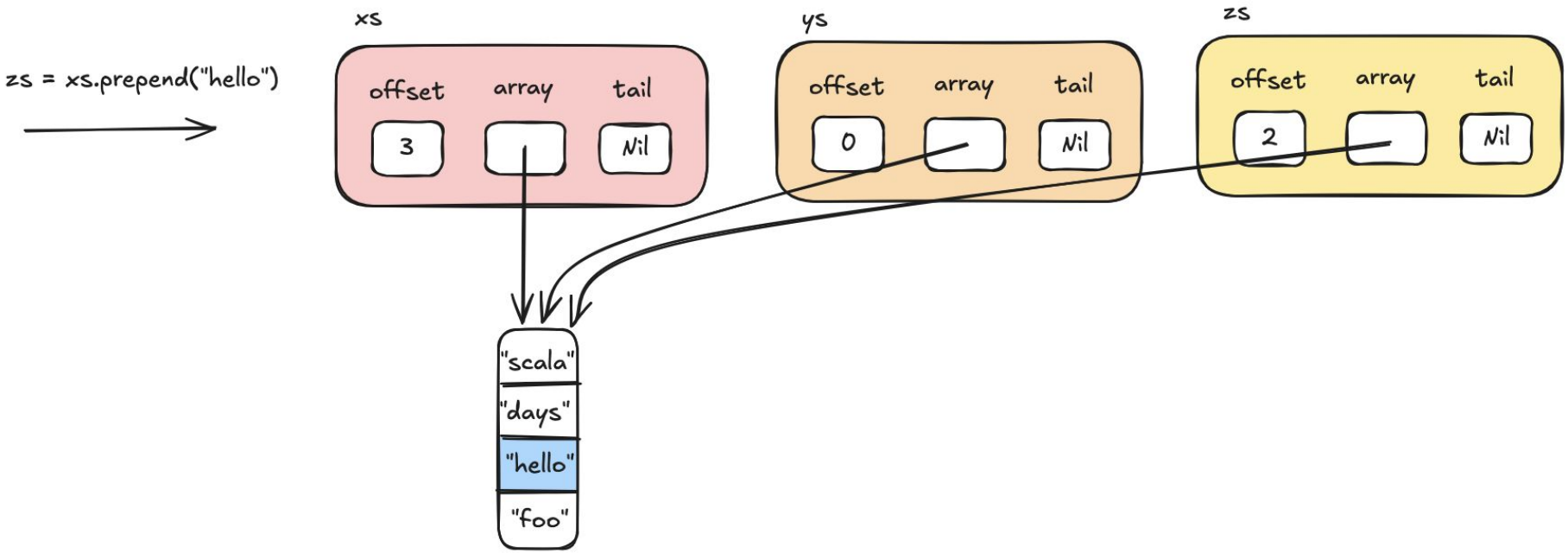
3. The mutability of the underlying array lets us make optimizations



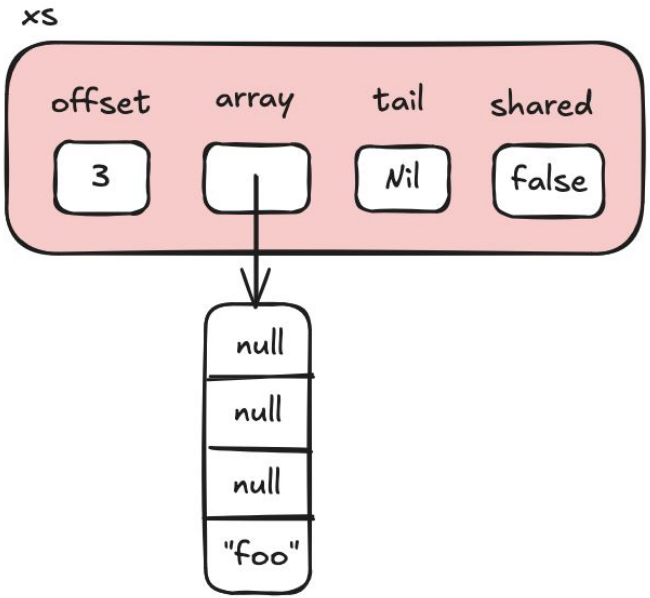
3. The mutability of the underlying array lets us make optimizations



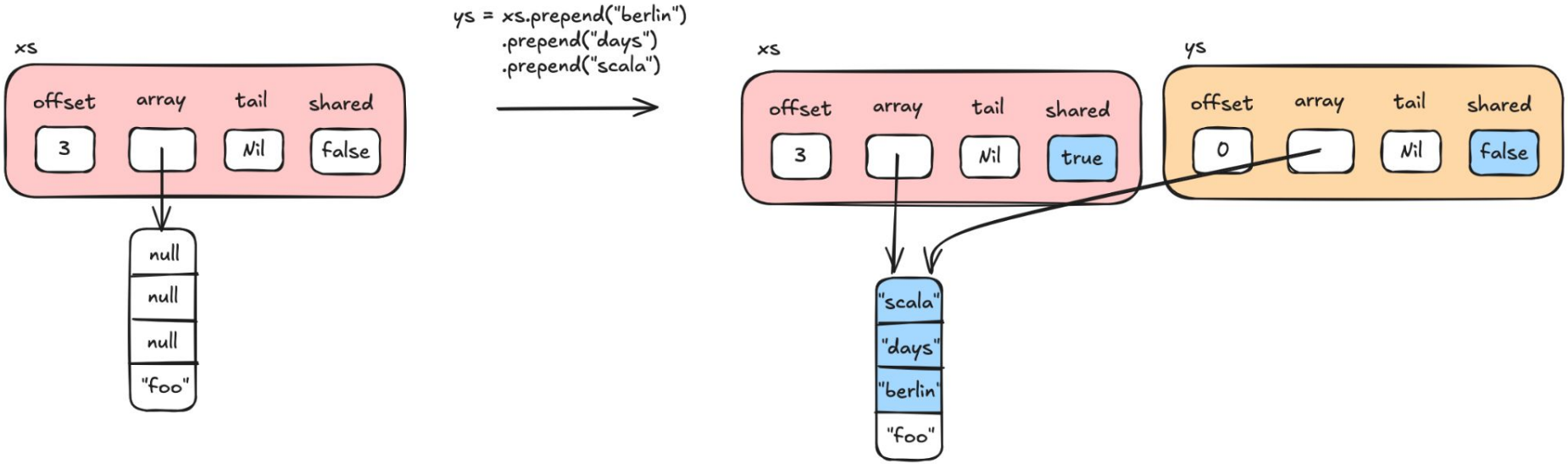
3. The mutability of the underlying array lets us make optimizations



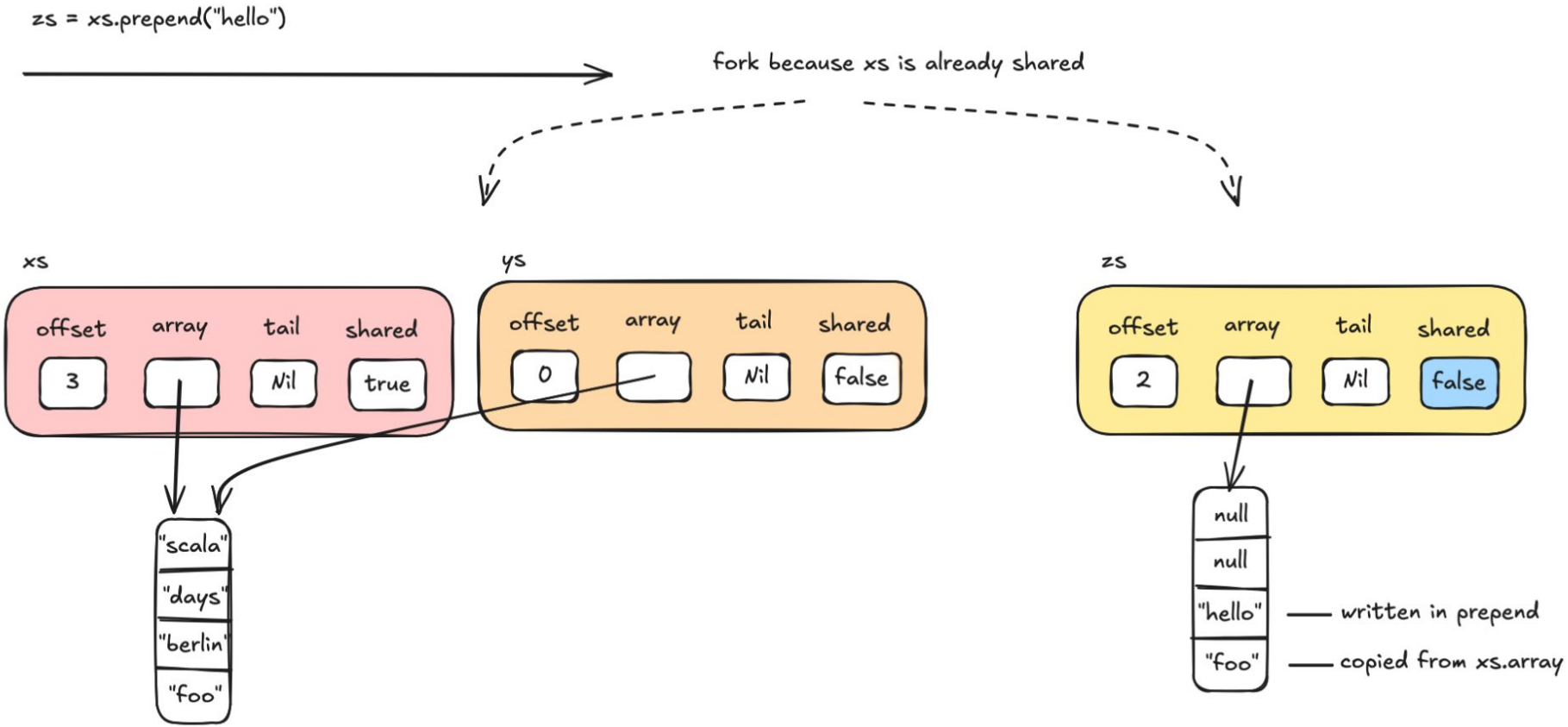
3. The mutability of the underlying array lets us make optimizations



3. The mutability of the underlying array lets us make optimizations



3. The mutability of the underlying array lets us make optimizations



3. The mutability of the underlying array lets us make optimizations

Prepend in a Concurrent Environment

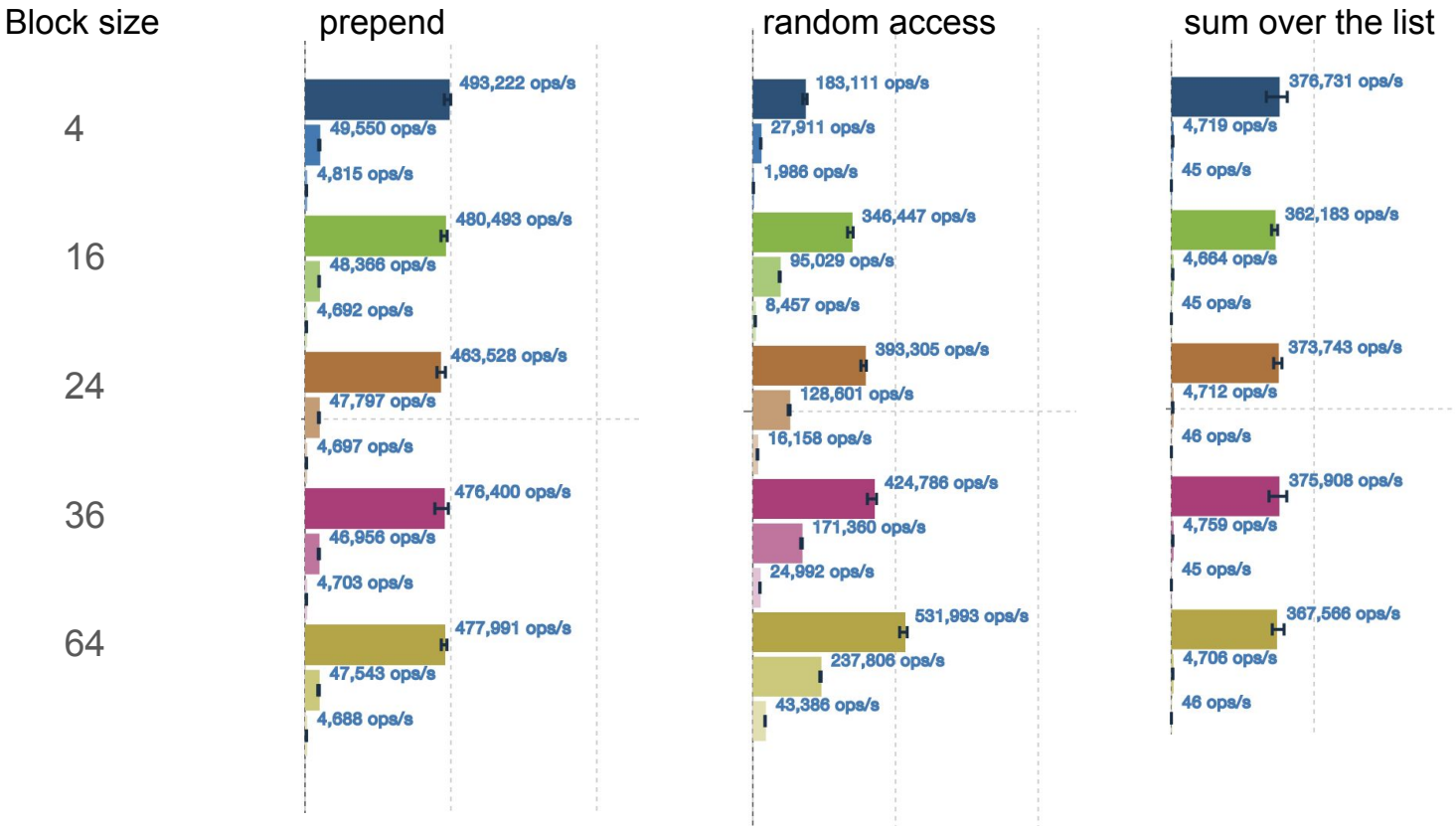
The *shared* field is an *Atomic* Boolean. This ensures memory synchronization in concurrent environments

```
def prepend[B >: A](a: B): BList.NonEmpty[B] = {
  if (offset > 0) {
    val nextOffset = offset - 1
    if (shared.compareAndSet(false, true)) { // not shared
      block(nextOffset) = a.asInstanceOf[A]
      Impl(nextOffset, block, tailBList)
    } else { // shared, forking detected
      val ary = new Array[Any](BlockSize)
      System.arraycopy(block, offset, ary, offset, BlockSize - offset)
      ary(nextOffset) = a
      Impl(nextOffset, ary, tailBList)
    }
  } else {
    val ary = new Array[Any](BlockSize)
    val offset = BlockSize - 1
    ary(offset) = a
    Impl(offset, ary, this)
  }
}
```

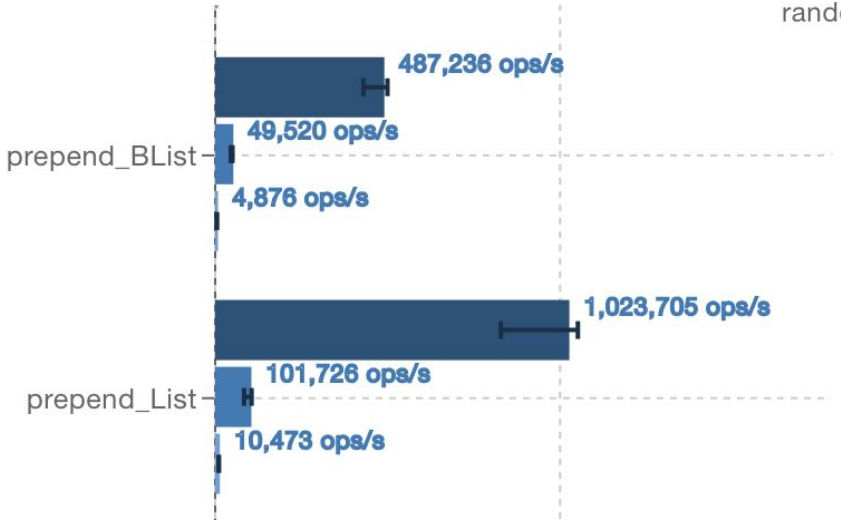
“We can make optimizations by taking advantage of the mutability of the underlying array”

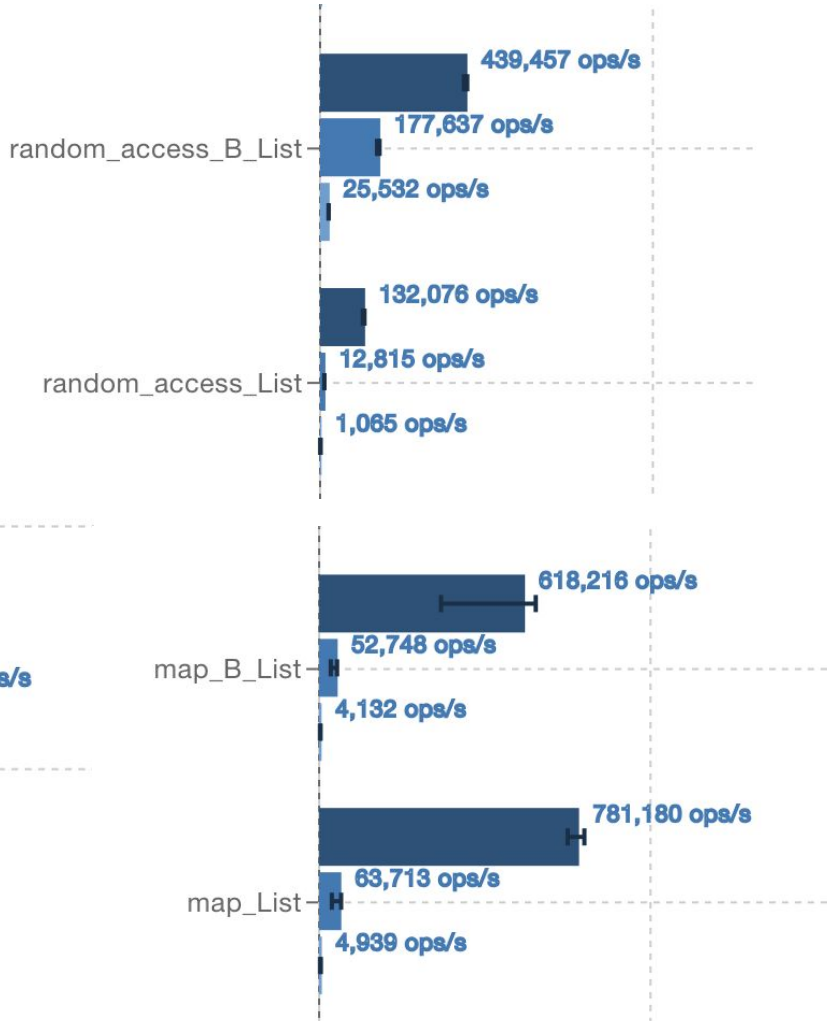
BList Results

Design decisions: determining the block size



Benchmarking





Benchmarking - BList Builder

- Important methods including map and fromList rely on BList's mutable Builder
- BList's Builder is slower than List's Builder in benchmarking - why?
  - BList overhead?
  - Advantage from batching allocations is not enough?
  - … something else?
- This stands in the way of being competitive with Scala's List

Work Completed and Where it Lives

cats-collections PR #837

- Methods: prepend, head, tail, get, map, filter, folds, splitAt, reverse, concat, drop, take, flatMap, forEach, toList, fromList, iterator, etc.
- Mutable Builder
- NonEmpty BList
- Type Classes: Eq, Alternative, Monad, Traverse
- Test suite with unit tests, property tests, and law tests

Thank you

A special thanks to Arman and Sergey for being amazing mentors throughout this project