Ha(r)sh Visitors

What's wrong with Swift's Hashable
and how to improve it.

Vincent Esche | @regexident
blog.definiteloops.com

What is Hashable?

And why do we need it to begin with?

Elevator Pitch:

“A hash function helps HashTables/ HashMaps find values efficiently and without touching unrelated elements.”

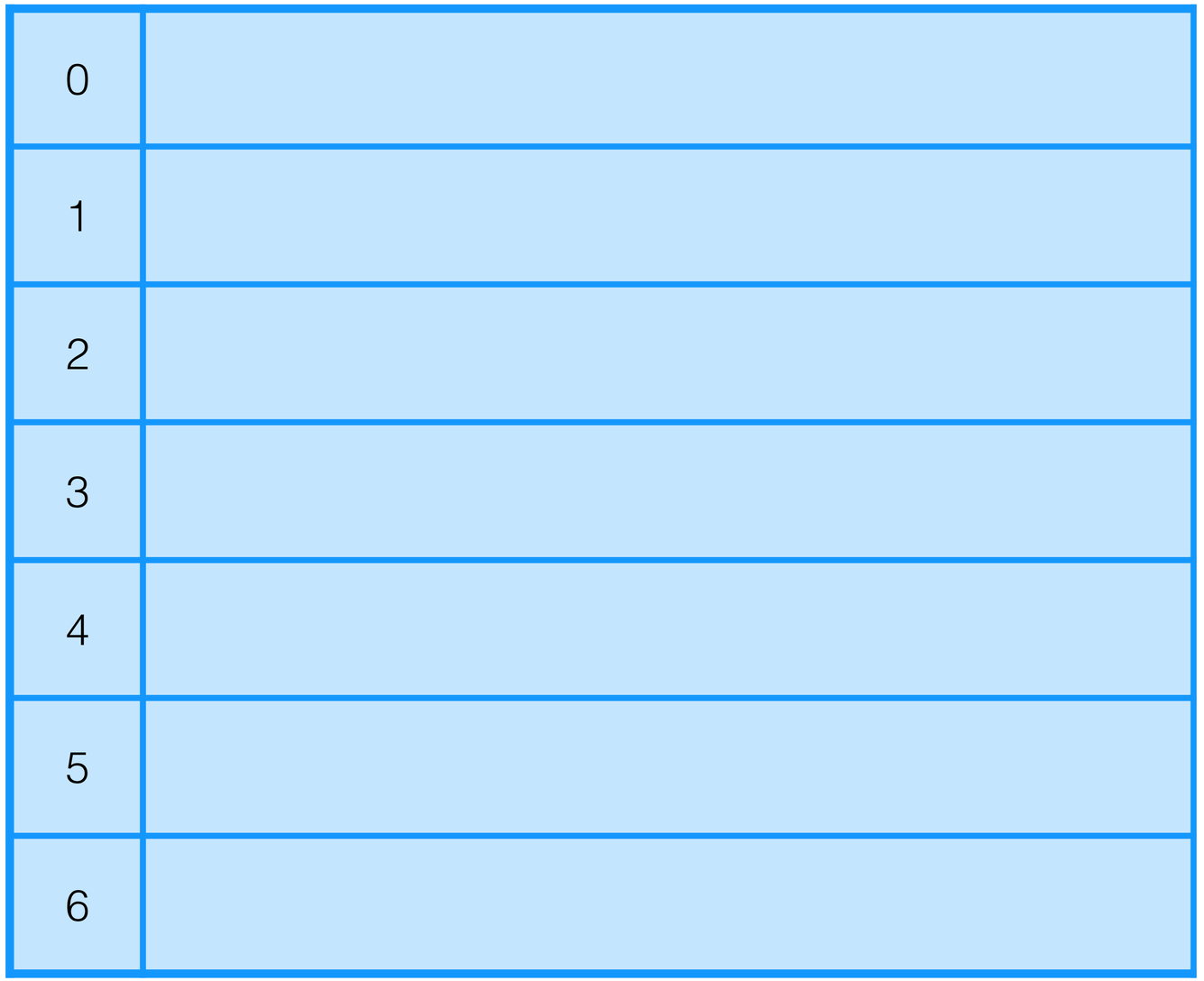
There’s quite a bit more to them, but it’s enough for now.

“Foobar”.hashValue = 3

self.insert(“Foobar”)

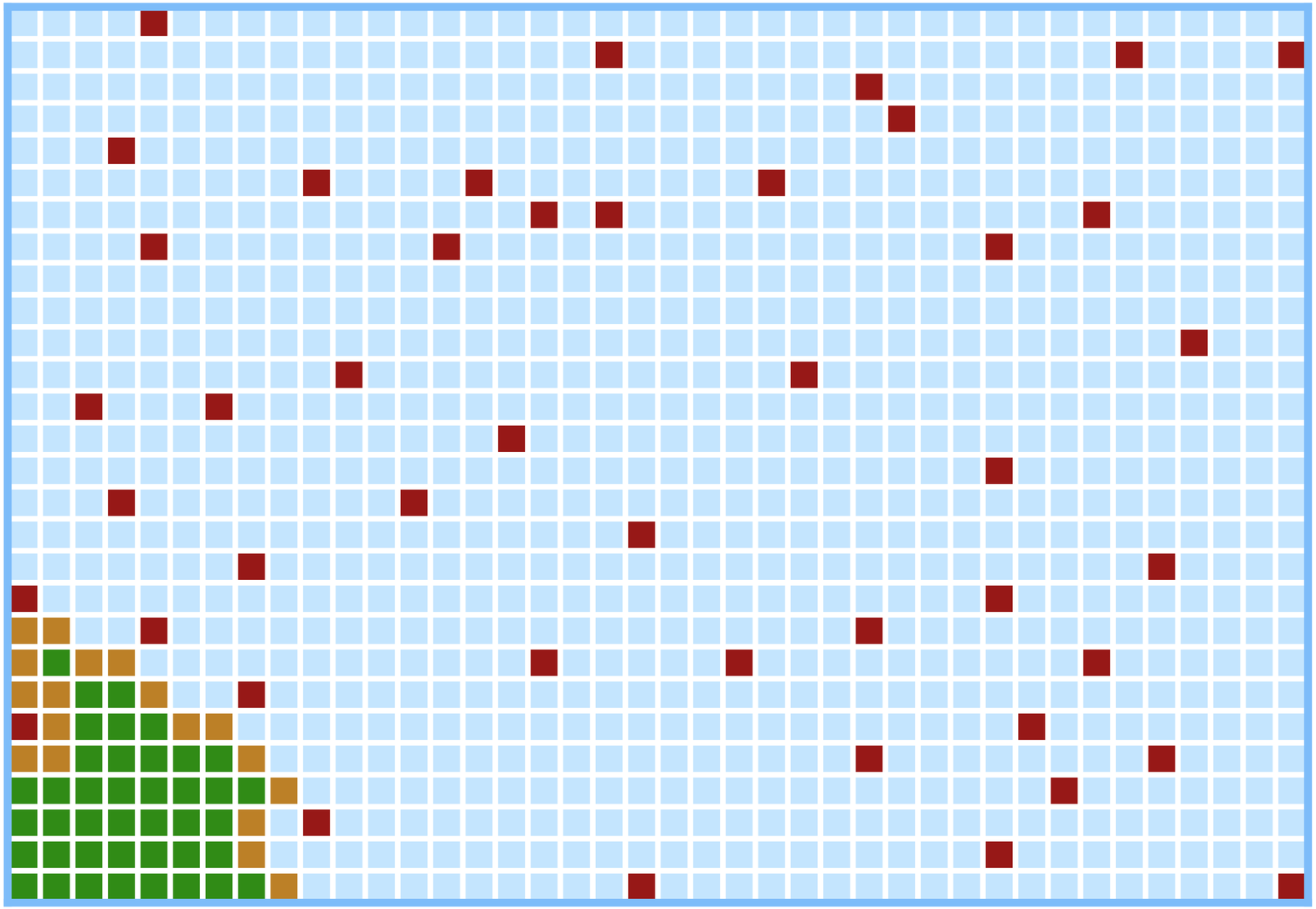
HashTable

// Gross over-simplification!



Using `Hashable`

Minesweeper



```
struct GridPoint {
  var x: Int
  var y: Int
}

class Minesweeper {
  let grid: (columns: Int, rows: Int)

  let mines: Set<GridPoint>
  var cleared: Set<GridPoint>
  var flagged: Set<GridPoint>
}
```

```
// There are way more efficient ways
// to code this using bitfields e.g.,
// but it'll do just fine for this demo.
```

```
error: type ‘GridPoint’ does not
 conform to protocol ‘Equatable’

error: type ‘GridPoint’ does not
 conform to protocol ‘Hashable’
```

```
struct GridPoint {
  var x: Int
  var y: Int
}

extension GridPoint: Equatable {
  static func == (
    lhs: GridPoint,
    rhs: GridPoint
  ) → Bool {
    return lhs.x == rhs.x && lhs.y == rhs.y
  }
}
```

```
error: type ‘GridPoint’ does not
 conform to protocol ‘Equatable’

error: type ‘GridPoint’ does not
 conform to protocol ‘Hashable’
```

What makes a Hash Function?

A hash function maps values from **variable sized input** to **fixed sized output values**, known as the hash values.

A hash function's **mapping behavior must be persistent** during the lifetime of a program.

A hash function may map **multiple input values** to a **single hash value**.

// Don't do this at home!

```
struct GridPoint {
  var x: Int
  var y: Int
}

extension GridPoint: Hashable {
  var hashValue: Int {
    return 42
  }
}
```

// Don't do this at home either!

```
struct GridPoint {
  var x: Int
  var y: Int
}

extension GridPoint: Hashable {
  var hashValue: Int {
    return x.hashValue ^ y.hashValue
  }
}
```

```
let (width, height) = (100, 100)
let total = width * height
var hashes = Set<Int>()
for x in 0 ..< width {
  for y in 0 ..< height {
    let gridPoint = GridPoint(x: x, y: y)
    hashes.insert(gridPoint.hashValue)
  }
}
print("\(hashes.count) unique hashes out of a
total of \(total).")
```

Demo!

That’s a 100% collision rate!

This is bad. Really bad.

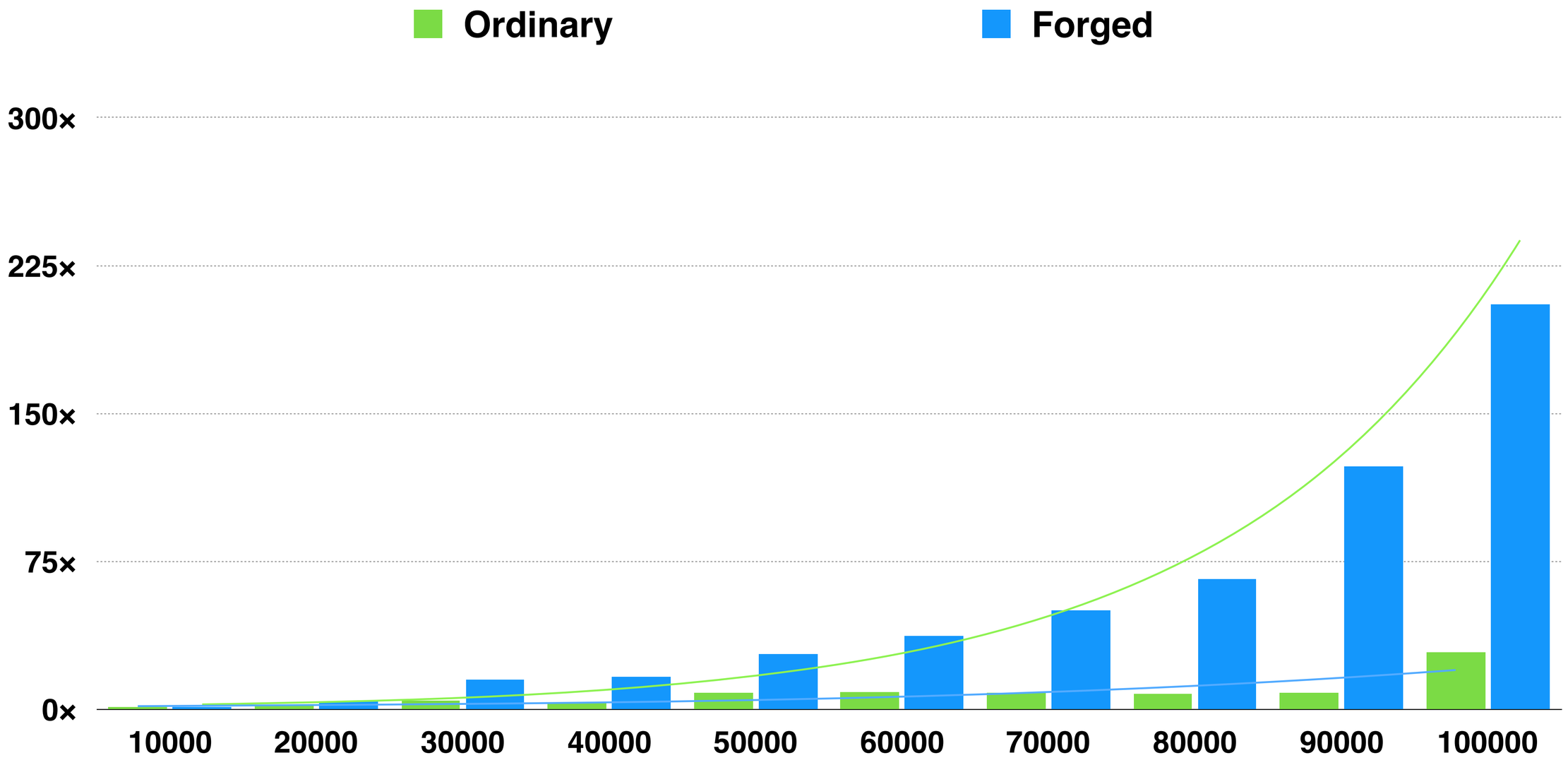
```
return x.hashValue ^ y.hashValue
```

≈

```
return 42
```

Give me some Numbers!

I mean, what’s wrong with some collisions?


Ordinary
Forged
300×
225×
150×
75×
0×
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000

What else is wrong with Hashable?

I mean, why not just fix the silly function, right?

Application developers should not have to implement hashing algorithms.

Have you ever tested your hash functions for quality?

Solution:

Don’t require users to implement hashing.

Valid implementations often require domain-knowledge.

String & Float/Double are really hard to hash properly.

Solution:

Don’t require users to implement hashing.

Badly implemented hashing opens the doors for DDOS attacks.

A language released in 2015 should have a solution for this.

Solution:

Provide secure hashing algorithms with randomized seeds.

One cannot exchange hashing algorithms.

The world of today is different from before the web.

Solution:

Allow users to choose the hashing algorithm being used.

One cannot use multiple hashing algorithms per type.

Be fast where it matters, secure everywhere else.

Solution:

Allow users to choose the hashing algorithm per instance.

How would we do this?

I mean, you said we could, didn’t you?

Remember NSCoder?

The Protocol, not the meetup! ;)

```
struct GridPoint {
   var x: Int
   var y: Int
}

extension GridPoint: NSCoding {
   func encode(with coder: NSCoder) {
      coder.encode(self.x, forKey: “x”)
      coder.encode(self.y, forKey: “y”)
   }
}
```

```
struct GridPoint {
  var x: Int
  var y: Int
}

extension GridPoint: Hashable {
  func hash(with hasher: inout Hasher) {
    hasher.hash(self.x)
    hasher.hash(self.y)
  }
}
```

What would make this work?

Spoiler: Remember the title, ”Ha(r)sh visitors”? ;)

```
protocol Hashable { // visitee
  func hash(with hasher: inout Hasher)
}

protocol Hasher { // visitor
  func finish() → Int
  func write(bytes: UnsafeRawBufferPointer)
}

protocol HasherBuilder { // helper
  func build() → Hasher
}
```

```
extension GridPoint: Hashable {
  func hash(with hasher: inout Hasher) {
    self.x.hash(with: &hasher)
    self.y.hash(with: &hasher)
  }
}

// Running the test from earlier with this
// API and a proper Hasher again, we get
// a collision rate of 0%! 😍
```

1. Application developers **do not have** to implement hashing algorithms. ✅

2. **No domain-knowledge required** for valid implementations. ✅

3. **Safe** from DDOS attacks. ✅

4. **Easily** exchange hashing algorithms.

5. **Easily** use **multiple** hashing algorithms per type.

```
struct Fnv1aHash {
  fileprivate var state: UInt
  init(seed: UInt) {
    self.state = seed &+ 14695981039346656037
  }
}
```

```
extension Fnv1aHash: Hasher {
  mutating func write(
    bytes: UnsafeRawBufferPointer
  ) {
    for byte in bytes {
      self.state = self.state ^ UInt(byte)
      self.state = self.state &* 1099511628211
    }
  }
  func finish() → Int {
    return Int(bitPattern: self.state)
  }
}
```

1. Application developers **do not have** to implement hashing algorithms. ✅

2. **No domain-knowledge required** for valid implementations. ✅

3. **Safe** from DDOS attacks. ✅

4. **Easily** exchange hashing algorithms. ✅

5. **Easily** use **multiple** hashing algorithms per type.

```
struct MySet<Element> where Element: Hashable {
  var buckets: [[Element?]] = []
  let hasherBuilder: HasherBuilder

  init() {
    self.init(
      hasherBuilder: DefaultHasherBuilder()
    )
  }

  init(hasherBuilder: HasherBuilder) {
    self.hasherBuilder = hasherBuilder
  }
}
```

1. Application developers **do not have** to implement hashing algorithms. ✅

2. **No domain-knowledge required** for valid implementations. ✅

3. **Safe** from DDOS attacks. ✅

4. **Easily** exchange hashing algorithms. ✅

5. **Easily** use **multiple** hashing algorithms per type. ✅

```
extension MySet {
  func contains(element: Key) → Bool {
    let index = self.bucket(for: element)
    let bucket = self.buckets[index]
    return bucket.contains(element)
  }

  func bucket(for element: Element) → Int {
    var hasher = self.hasherBuilder.build()
    element.hash(with: &hasher)
    return hasher.finish() % self.capacity
  }
}
```

```
extension Hashable {
  func hash(with hasher: inout Hashable) {
    self.hashValue.hash(with: &hasher)
  }
}

// `var hashValue` would be declared
// deprecated and implementing it be made
// to emit a compiler warning urging
// a migration to `func hash(with:)`.
```

```
var cocoaHeadsTeam: MySet(
  hasherBuilder: FastHasherBuilder()
)

cocoaHeadsTeam.formUnion([
  “Lukasz", “Martin”, “Melissa”, “Reiner”
])

team.remove(“Reiner”)

team.insert(“Vincent”)

// “Lukasz”, “Martin”, “Melissa”, “Vincent”
```

1. Application developers **do not have** to implement hashing algorithms. ✅

2. **No domain-knowledge required** for valid implementations. ✅

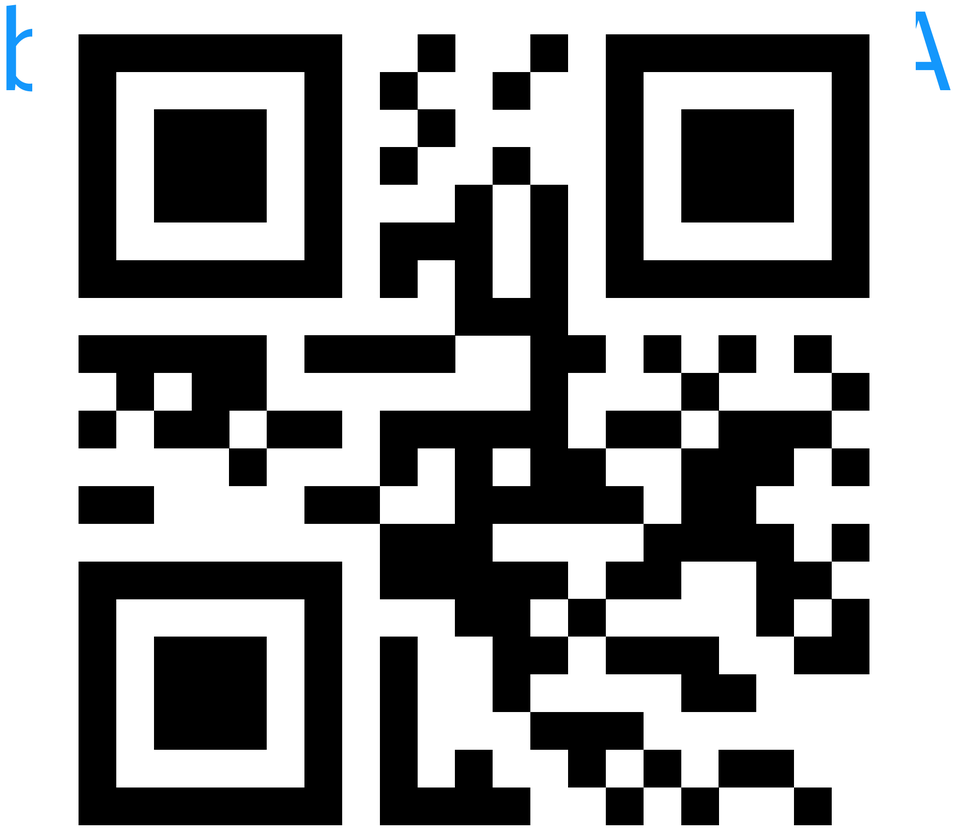
3. **Safe** from DDOS attacks. ✅

4. **Easily** exchange hashing algorithms. ✅

5. **Easily** use **multiple** hashing algorithms per type. ✅

Demo!

Wanna know more?



Thanks!

Questions?

Vincent Esche | @regexident
blog.definiteloops.com