Inside the Map Implementation

Keith Randall
@GopherCon, 2016/07/12

Can’t cover it all in ½ hour!

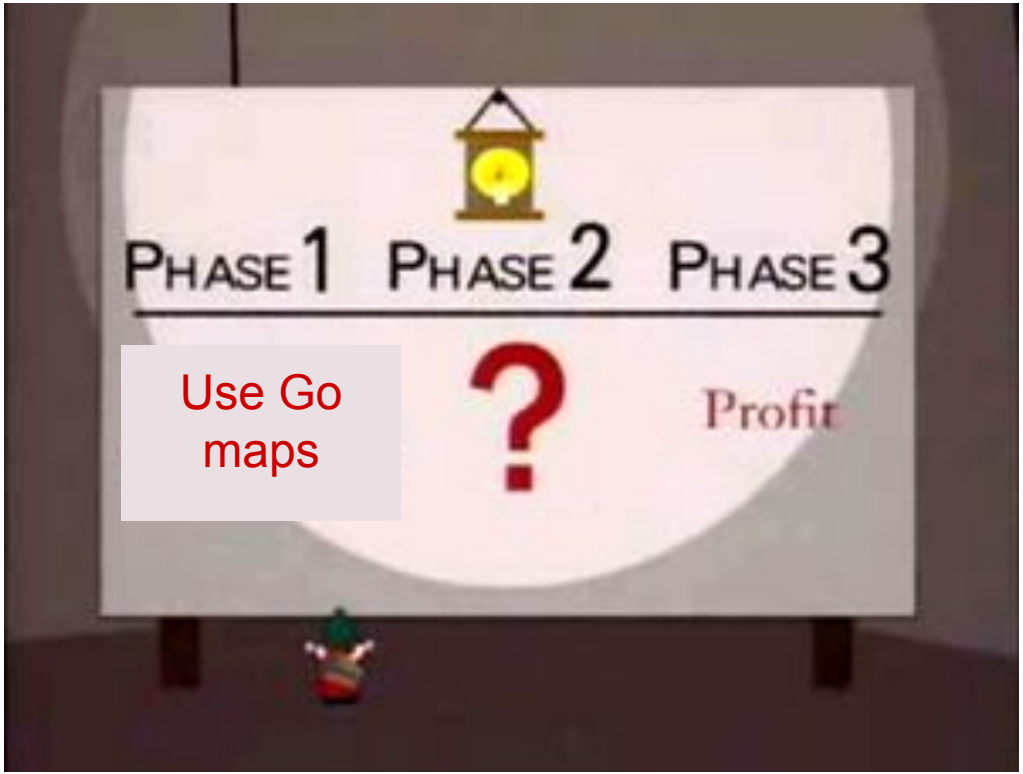
What are maps?

Associative containers mapping keys to values.

- Construct: `m := map[key]value{}`
- Insert: `m[k] = v`
- Lookup: `v = m[k]`
- Delete: `delete(m, k)`
- Iterate: `for k, v := range m`
- Size: `len(m)`

Key type must have an == operation (no maps, slices, or funcs as keys).

Operations run in constant expected time.


PHASE 1
PHASE 2
PHASE 3
Use Go maps
?
Profit


PHASE 1
PHASE 2
PHASE 3
Use Go maps
Stock Trading
Profit

High-frequency trading bot

```
var nasdaq chan struct {
     symbol string
     price  float64
}
m := map[string]float64{}
for x := range nasdaq {
     last := m[x.symbol]
     if x.price > last {
          fmt.Printf("buy %s!\n", x.symbol)
     }
     if x.price < last {
          fmt.Printf("sell %s!\n", x.symbol)
     }
     m[x.symbol] = x.price
}
```

NASDAQ stock prices

MSFT	50.81
AAPL	94.56
GOOG	706.63
CSCO	26.72
ORCL	39.47
INTC	29.99
VOD	34.13
QCOM	52.79
AMZN	697.45
AMGN	150.83
…	

High-frequency trading bot

```
var nasdaq chan struct {
     symbol string
     price  float64
}
m := map[string]float64{}
for x := range nasdaq {
     last := m[x.symbol]
     if x.price > last {
          fmt.Printf("buy %s!\n", x.symbol)
     }
     if x.price < last {
          fmt.Printf("sell %s!\n", x.symbol)
     }
     m[x.symbol] = x.price
}
```

NASDAQ stock prices

MSFT	50.81
AAPL	94.56
GOOG	706.63
CSCO	26.72
ORCL	39.47
INTC	29.99
VOD	34.13
QCOM	52.79
AMZN	697.45
AMGN	150.83
…	

High-frequency trading bot

```
var nasdaq chan struct {
      symbol string
      price  float64
}
m :=  o[string]float64{}
for      nasdaq {
         x.symbol]
      if x.    > last {
           Printf("buy %s!\n", x.symbol)
      }
      if x.price < last {
           fmt.Printf("sell %s!\n", x.symbol)
      }
      m[x.symbol] = x.price
}
```

NASDAQ stock prices

MSFT	50.81
AAPL	94.56
GOOG	706.63
CSCO	2 72
ORCL	7
INTC	2 9
V	4.13
Q	2.79
AM	697.45
AMGN	150.83
…	

High-frequency trading bot

```
var nasdaq chan struct {
	symbol string
	price  float64
}
m :[illegible]p[string]float6[illegible]
for[illegible] nasdaq
	[illegible]x.symbol]
	if x.[illegible]e > la[illegible]{
		[illegible]Printf[illegible]%[illegible]\n”, x.sym[illegible]l)
	}
	if x.price < last {
		fmt.Print[illegible]“sell %s!\n”, x.symbol)
	}
	m[x.symbol][illegible]x.price
}
```

NASDAQ stock prices

MSFT	50.81
AAPL	94.56
GOOG	706.63
CSCO	[illegible]72
ORCL	[illegible]7
INTC	2[illegible]9
V[illegible]	[illegible]4.13
QC[illegible]	[illegible]2.79
AM[illegible]	[illegible]697.45
AMGN	150.83
…	

High-frequency trading bot

Simple implementation

```
type entry struct {
   k string
   v float64
}
type map []entry
func lookup(m map, k string) float64 {
    for _, e := range m {
        if e.k == k {
            return e.v
        }
    }
    return 0
}
```

Simple implementation

```
type entry struct {
   k string
   v float64
}
type map []entry
func lookup(m map, k string) float64 {
     for _, e := range m {
          if e.k == k {
               return e.v
          }
     }
     return 0
}
```

Speed Fail!

Idea

Idea: split data up into buckets!

A-F		G-M		N-S		T-Z	
AAPL	94.56	MSFT	50.81	ORCL	39.47	VOD	34.13
CSCO	26.72	GOOG	706.63	QCOM	52.79		
AMZN	697.45	INTC	29.99				
AMGN	150.83						

```
bucket = (symbol[0]-'A')*4/('Z'-'A')
```

Idea: split data up into buckets!

A-F		G-M		N-S		T-Z	
AAPL	94.56	MSFT	50.81	ORCL	39.47	VOD	34.13
CSCO	26.72	GOOG	706.63	QCOM	62.79		
AMZN	697.45	INTC	29.99				
AMGN	150.83						

```
bucket = (symbol[0]-'A')*4/('Z'-'A')
```

Speed Fixed!

URLDAQ

A-F	G-M	N-S	T-Z
	http://google.com http://intel.com http://cisco.com http://apple.com http://amazon.com http://amgen.com http://microsoft.com http://oracle.com http://qualcomm.com http://vodafone.com		

URLDAQ

Bucketing by first letter isn't very good.
Is there a way to reliably split the data into buckets?

Hash function

Choose a bucket for each key so that entries are distributed as evenly as possible.

```
bucket = h(key)
```

Required for correctness:

- Deterministic: `k1 == k2` ⟶ `h(k1) == h(k2)`

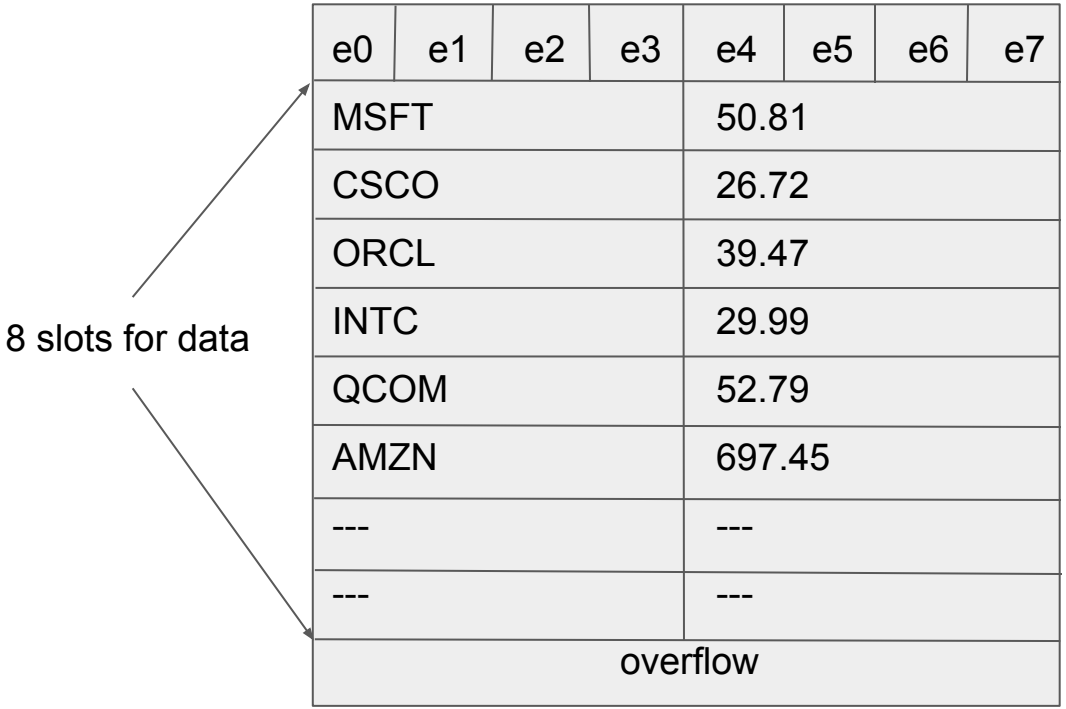
Want for performance:

- Uniform: Probability[`h(k) == b`] ~= 1/#buckets
- Fast: `h(k)` can be computed quickly
- Adversary safe: hard for attacker to find lots of `k` with `h(k) == b`

Good news: Such hash functions exist.
Bad news: Hard to get right.

Go gets it right for you. ⟶ no user-defined hash functions.

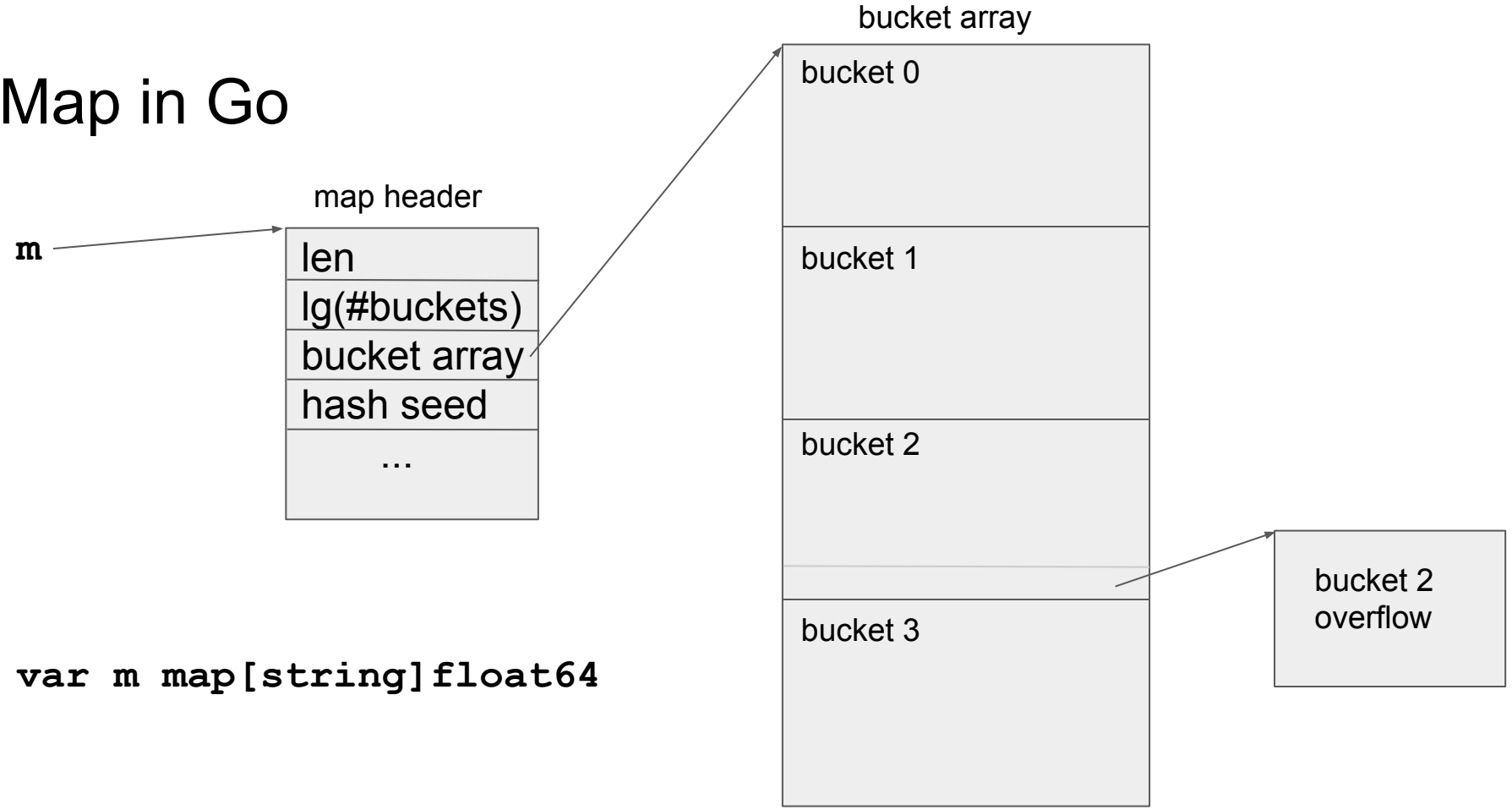
Map bucket in Go



e_i = 8-bit slots for extra hash bits

overflow points to another bucket if we need more than 8 slots.

Map in Go




```
var m map[string]float64
```

Lookup

```
v = m[k]
```

compiles to

```
v = runtime.lookup(m, k)
```

Lookup

```
v = m[k]
```

compiles to

```
v = runtime.lookup(m, k)
```

where the runtime has the function

```
func<K,V> lookup(m map[K]V, k K) V
```

Lookup

```
v = m[k]
```

compiles to

```
v = runtime.lookup(m, k)
```

where the runtime has the function

```
func<K,V> lookup(m map[K]V, k K) V
```

Generics Fail!

Faking generics

- All operations are done using **`unsafe.Pointer`**s pointing to values of generic type.
- Type information for each value is handled by a ***type descriptor***.
- Type descriptors provide operations like ==, hash, copy.

```
type _type struct {
    size  uintptr
    equal func(unsafe.Pointer, unsafe.Pointer) bool
    hash  func(unsafe.Pointer, uintptr) uintptr
    ...
}
type mapType struct {
    key   *_type
    value *_type
    ...
}
```

Lookup

```
v = m[k]
```

compiles to

```
pk := unsafe.Pointer(&k)
pv := runtime.lookup(typeOf(m), m, pk)
v = *(*V)pv
```

where the runtime has the function

```
func lookup(t *mapType,
            m *mapHeader,
            k unsafe.Pointer) unsafe.Pointer
```

Lookup implementation, page 1 of 2

```
// lookup looks up a key in a map and returns a pointer to the associated value.
// t = type of the map
// m = map
// key = pointer to key
func lookup(t *mapType, m *mapHeader, key unsafe.Pointer) unsafe.Pointer {
     if m == nil || m.count == 0 {
          return zero
     }

     hash := t.key.hash(key, m.seed)                          // hash := hashfn(key)
     bucket := hash & (1<<m.logB-1)                           // bucket := hash % nbuckets
     extra := byte(hash >> 56)                                // extra := top 8 bits of hash
     b := (*bucket)(add(m.buckets, bucket*t.bucketsize))      // b := &m.buckets[bucket]
```

Lookup implementation, page 2 of 2

```
    for {
        for i := 0; i < 8; i++ {
            if b.extra[i] != extra {                    // check 8 extra hash bits
                continue
            }
            k := add(b, dataOffset+i*t.key.size)        // pointer to k_i in bucket
            if t.key.equal(key, k) {
                // return pointer to v_i
                return add(b, dataOffset+8*t.key.size+i*t.value.size)
            }
        }
        b = b.overflow
        if b == nil {
            return zero
        }
    }
}
```

Growing the map

When buckets get too full, we need to grow the map.

“too full” = average of 6.5 entries per bucket

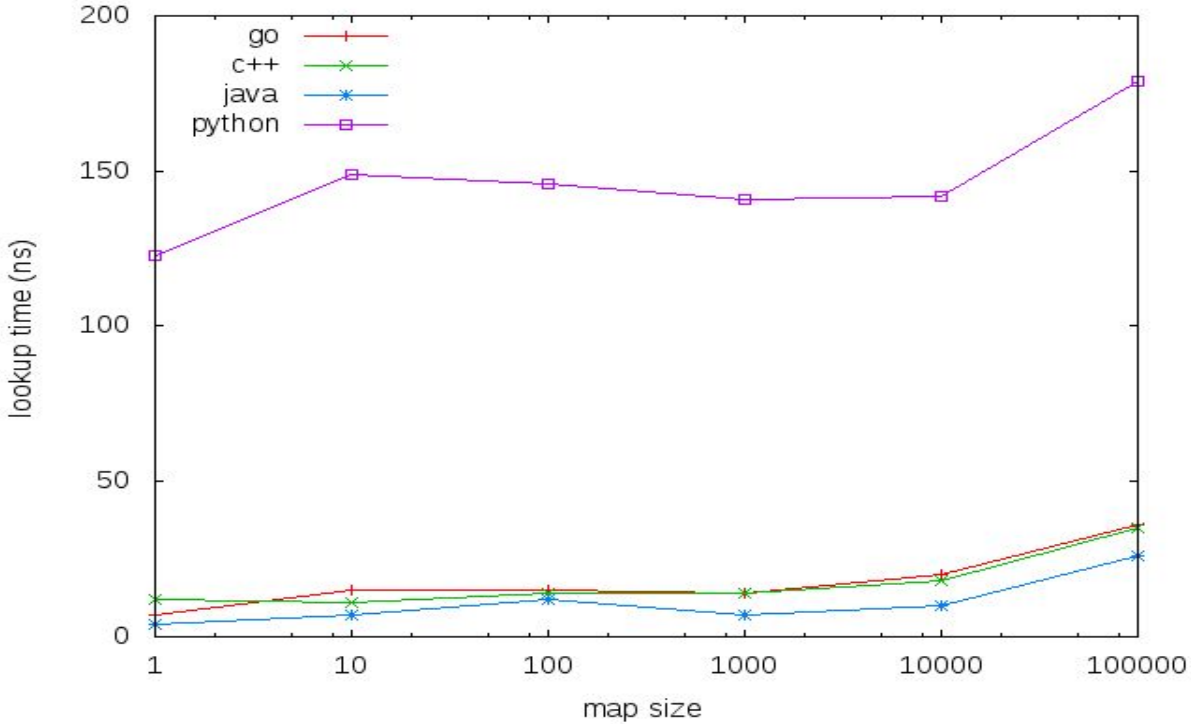
1. Allocate a new array of buckets of twice the size
2. Copy entries over from the old buckets to the new buckets
3. Use the new buckets

   The process of copying is done incrementally, a little bit during each insert or delete. During copying, operations on the map are a bit more expensive.

Maps in other languages

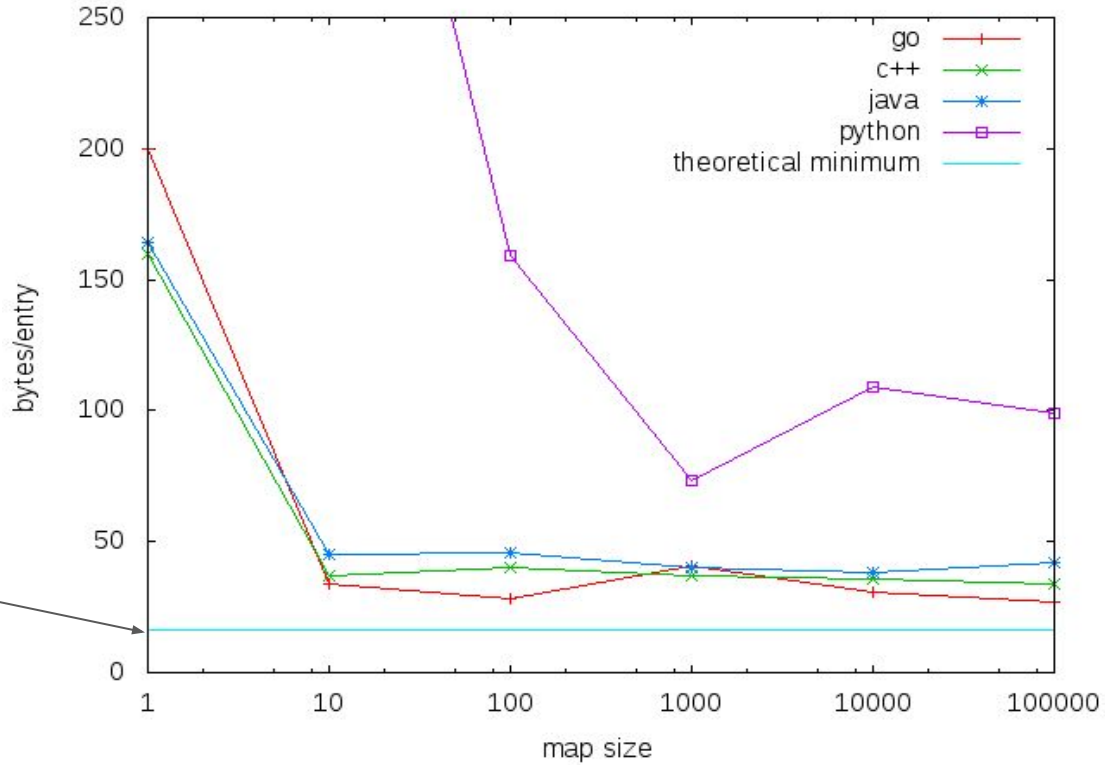
	c++	Java	Python	Go
&m[k] allowed	yes	no	no	no
Modify during iteration	no	no	no	yes
Provide your own ==, hash	yes	yes	yes	no
Adversary safe	no	yes*	no+	yes

Speed


200
150
100
50
0
lookup time (ns)
go
c++
java
python
1
10
100
1000
10000
100000
map size

Space

map[int64]int64
16 bytes/entry



Conclusions

Use maps!

- A versatile data structure for the 21st century
- 2x space overhead compared to the raw data
- Access times on the order of ~100 cycles

Conclusions

Use maps!

- A versatile data structure fo[illegible]t century
- 2x space overhead compare[illegible]he raw data
- Access times on the order of ~[illegible]ycles

Conclusions

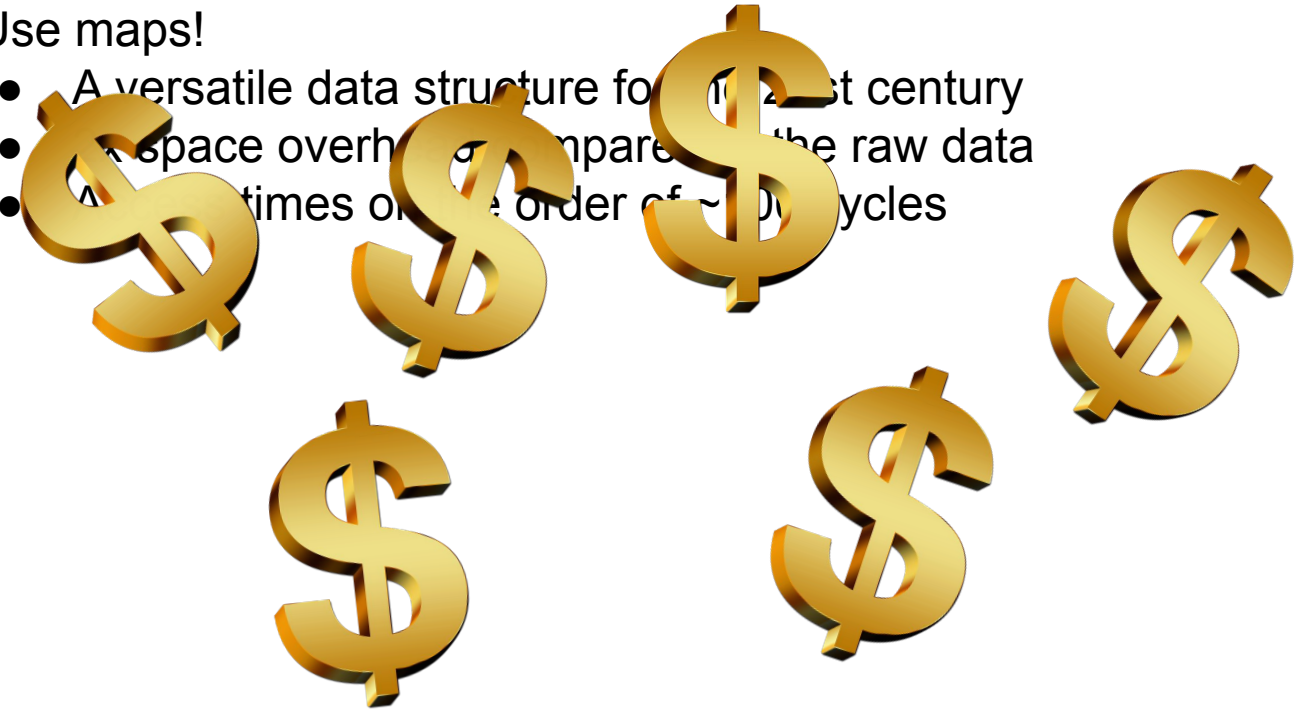
Use maps!

- A versatile data structure fo[illegible]t century
- [illegible] space overhead compare[illegible]he raw data
- A[illegible]times on the order of ~[illegible]ycles

Conclusions

Use maps!

- A versatile data structure fo[illegible]t century
- [illegible]space overh[illegible]mpare[illegible]he raw data
- A[illegible]times o[illegible]e order of ~[illegible]ycles



Conclusions

Use maps!

- A versati[illegible]ta[illegible]tructure fo[illegible]t ce[illegible]
- [illegible]space o[illegible]mpare[illegible]the raw d[illegible]
- [illegible]order of ~[illegible]cycle[illegible]

Properties

- Lookups typically do
  - 1 key hash
  - touch 1-3 consecutive cache lines
  - 1 key ==
- Keeping 8 items in a bucket
  - Reduces space overhead of overflow pointer
  - Can pack data without padding (e.g. map[int8]int64)
  - Touching a cache line anyway, might as well use all of it

Iterators

Unlike most languages, Go provides definite semantics of iterators in the presence of inserts and deletes.

Still to do

- m[k] += …
  - Currently requires 2 map operations, could do it with 1.
- Map shrinking
  - Has tricky interactions with iteration
- Do incremental grow work during reads
  - Currently a built but then static map has a chance of being forever in a growing state
  - Requires tricky synchronization
- Hash function implementation improvements
- Smaller overflow buckets

Insider tips

- Preallocate buckets if size known: `make(map[int]int, 1000)`
- There are special fast paths for `int32`, `int64`, and `string` keys.
- Clear a map: `for k := range m { delete(m, k) }`
- No allocation for `var b []byte; v := m[string(b)]`
- No pointers in key and value -> faster GC scanning