Adventures in Optimization

David Golden • @xdg
NY.pm • July 2014

The problem…

Perl hashes are unordered maps

Perl hashes are **random** unordered maps

```
$ perl -wE 'my %h = 1 .. 10; say "$_ => $h{$_}" for keys %h'
```

```
Perl 5.16           Perl 5.18

1 => 2              5 => 6
3 => 4              9 => 10
7 => 8              7 => 8
9 => 10             3 => 4
5 => 6              1 => 2

1 => 2              7 => 8
3 => 4              3 => 4
7 => 8              5 => 6
9 => 10             1 => 2
5 => 6              9 => 10

1 => 2              9 => 10
3 => 4              1 => 2
7 => 8              3 => 4
9 => 10             7 => 8
5 => 6              5 => 6
```

```
$ perl -wE 'my %h = 1 .. 10; say "$_ => $h{$_}" for keys %h'
```

```
Perl 5.16       Perl 5.18

1 => 2          5 => 6
3 => 4          9 => 10
7 => 8          7 => 8
9 => 10         3 => 4
5 => 6          1 => 2

1 => 2          7 => 8
3 => 4          3 => 4
7 => 8          5 => 6
9 => 10         1 => 2
5 => 6          9 => 10

1 => 2          9 => 10
3 => 4          1 => 2
7 => 8          3 => 4
9 => 10         7 => 8
5 => 6          5 => 6
```

```
$ perl -wE 'my %h = 1 .. 10; say "$_ => $h{$_}" for keys %h'
```

```
Perl 5.16          Perl 5.18

1 => 2             5 => 6
3 => 4             9 => 10
7 => 8             7 => 8
9 => 10            3 => 4
5 => 6             1 => 2

1 => 2             7 => 8
3 => 4             3 => 4
7 => 8             5 => 6
9 => 10            1 => 2
5 => 6             9 => 10

1 => 2             9 => 10
3 => 4             1 => 2
7 => 8             3 => 4
9 => 10            7 => 8
5 => 6             5 => 6
```

What if order matters?

```
# MongoDB
$db->run_command(
  { insert => $collection, … }
);
```

```
# some web apps
http://example.com/?p1=one&p2=two
```

Order isn’t free

- Arrays of pairs — no quick random access
- Objects — method call overhead
- Tied hashes — tie + method overhead

Tie::IxHash?

```
# Tie interface
$t = tie( %myhash, ‘Tie::IxHash’,
   first => 1, second => 2
);
$myhash{third} = 3;
say $myhash{first};


# OO interface
$t = Tie::IxHash->new(
   first => 1, second => 2
);
$t->Push(third => 3);
say $t->FETCH(‘third’);
```

Tie::IxHash problems

- tied → very slow
- OO → ugly (“FETCH”)
- OO → expensive copy
- OO → no iterator

Maybe I could patch it

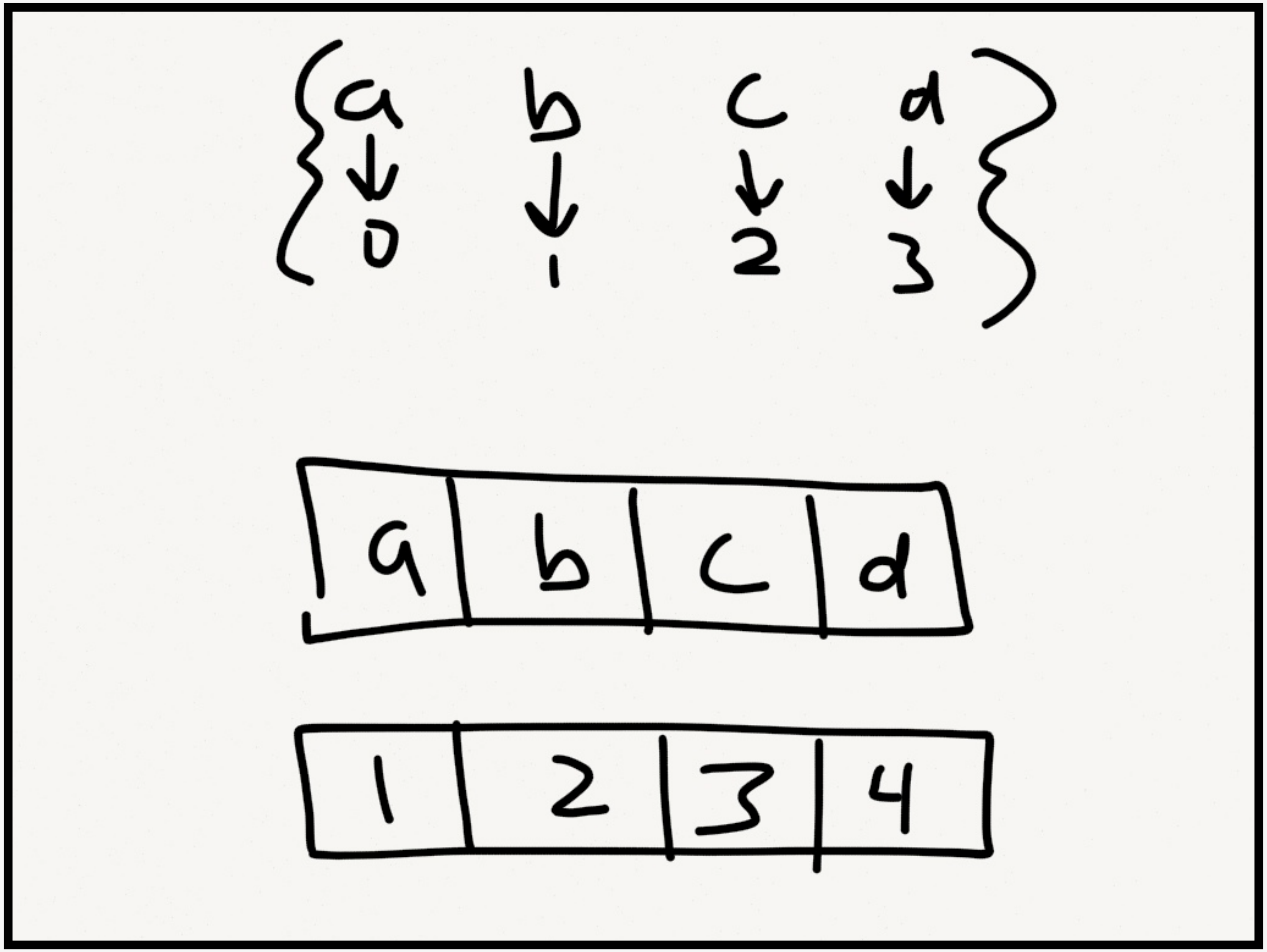
Tie::IxHash guts

```
sub TIEHASH {
  my($c) = shift;
  my($s) = [];
  $s->[0] = {};   # hashkey index
  $s->[1] = [];   # array of keys
  $s->[2] = [];   # array of data
  $s->[3] = 0;    # iter count

  bless $s, $c;

  $s->Push(@_) if @_;

  return $s;
}
```

WTF???

```
sub TIEHASH {
  my($c) = shift;
  my($s) = [];
  $s->[0] = {};    # hashkey index
  $s->[1] = [];    # array of keys
  $s->[2] = [];    # array of data
  $s->[3] = 0;     # iter count

  bless $s, $c;

  $s->Push(@_) if @_;

  return $s;
}
```

```
Tie::IxHash->new( a=>1, b=>2, c=>3, d=>4 );
```



Expensive fetch

```
sub FETCH {
  my($s, $k) = (shift, shift);
  return exists( $s->[0]{$k} ) ? $s->[2][ $s->[0]{$k} ] : undef;
}
```

- exists call
- ternary op
- 6 dereferences!

Expensive store

```
sub STORE {
  my($s, $k, $v) = (shift, shift, shift);

  if (exists $s->[0]{$k}) {
    my($i) = $s->[0]{$k};
    $s->[1][$i] = $k;
    $s->[2][$i] = $v;
    $s->[0]{$k} = $i;
  }
  else {
    push(@{$s->[1]}, $k);
    push(@{$s->[2]}, $v);
    $s->[0]{$k} = $#{$s->[1]};
  }
}
```

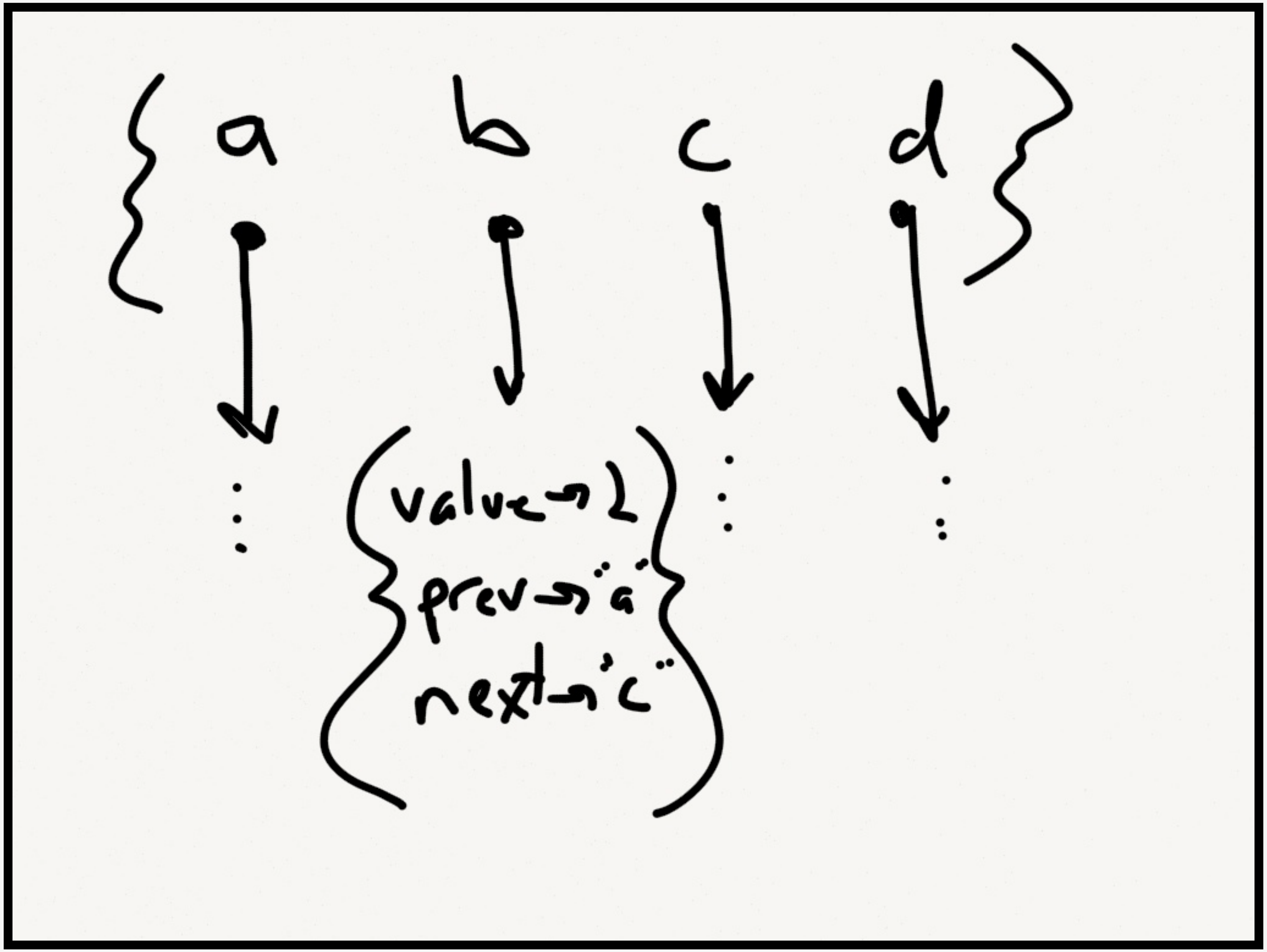
Anyone notice this?

```
sub STORE {
  my($s, $k, $v) = (shift, shift, shift);

  if (exists $s->[0]{$k}) {
    my($i) = $s->[0]{$k};
    $s->[1][$i] = $k;
    $s->[2][$i] = $v;
    $s->[0]{$k} = $i;
  }
  else {
    push(@{$s->[1]}, $k);
    push(@{$s->[2]}, $v);
    $s->[0]{$k} = $#{$s->[1]};
  }
}
```

Alternatives?

Tie::LLHash

```
tie %h, "Tie::LLHash", a=>1, b=>2, c=>3, d=>4;
```

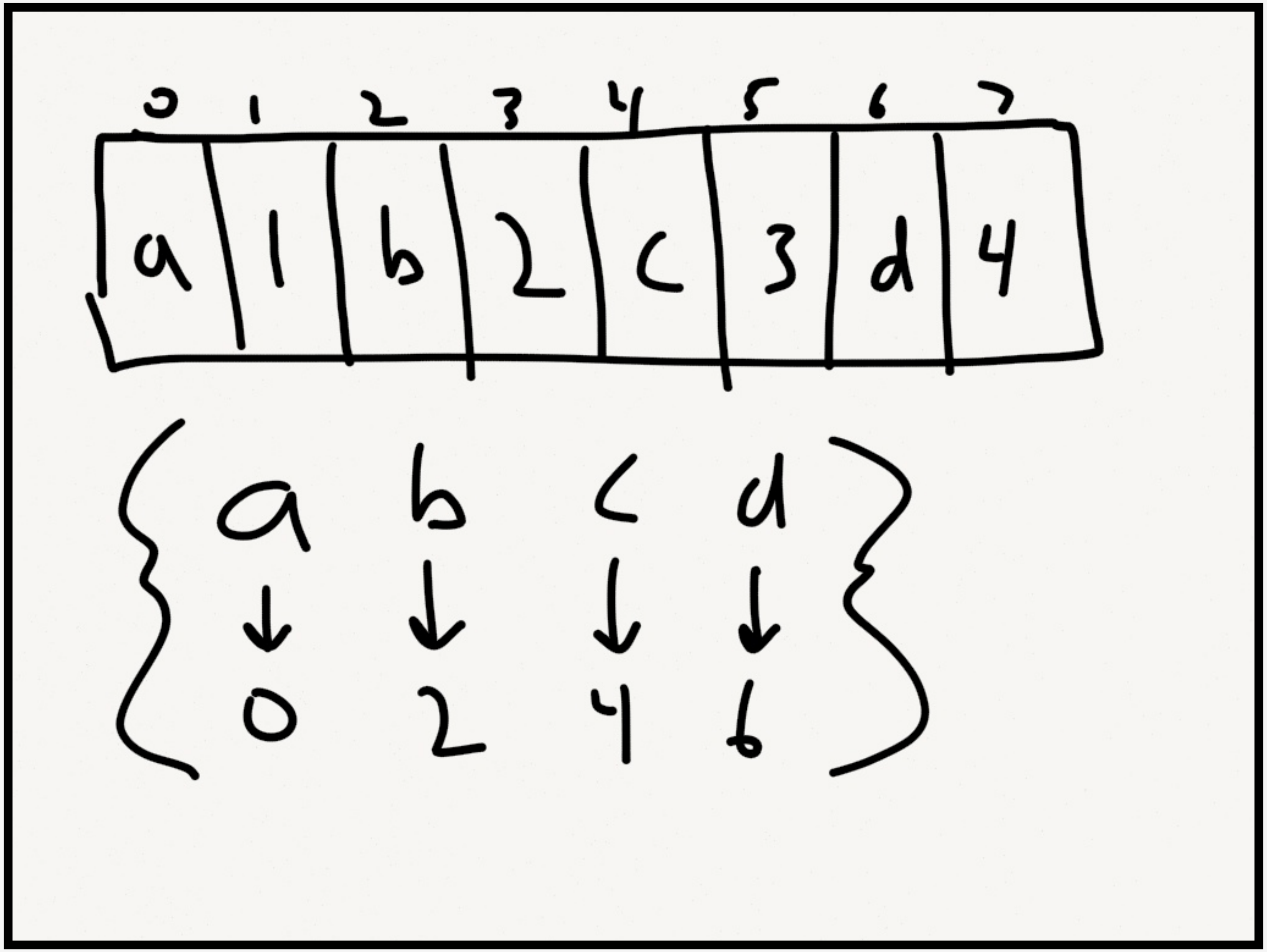


Memory allocation per key!

```
sub last {
   my $self = shift;

   if (@_) { # Set it
      my $newkey = shift;
      my $newvalue = shift;

      croak ("'$newkey' already exists") if $self->EXISTS($newkey);

      # Create the new node
      $self->{'nodes'}{$newkey} =
      {
         'next'  => undef,
         'value' => $newvalue,
         'prev'  => undef,
      };

      # Put it in its relative place
      if (defined $self->{'last'}) {
         $self->{'nodes'}{$newkey}{'prev'} = $self->{'last'};
         $self->{'nodes'}{ $self->{'last'} }{'next'} = $newkey;
      }

      # Finally, make this node the last node
      $self->{'last'} = $newkey;

      # If this is an empty hash, make it the first node too
      $self->{'first'} = $newkey unless (defined $self->{'first'});
   }
```

Array::AsHash

```
Array::AsHash->new({array =>[a=>1,b=>2,c=>3,d=>4]});
```



Subroutine call per key!

```
sub get {
    my ( $self, @keys ) = @_;
    my @get;
    foreach my $key (@keys) {
        $key = $self->$_actual_key($key);
        next unless defined $key;
        my $exists = $self->exists($key);
        if ( $self->{is_strict} && !$exists ) {
            $self->$_croak("Cannot get non-existent key ($key)");
        }
        if ($exists) {
            CORE::push @get, $self->{array_for}[ $self->$_index($key) + 1 ];
        }
        elsif ( @keys > 1 ) {
            CORE::push @get, undef;
        }
        else {
            return;
        }
    }
    return wantarray ? @get
      : @keys > 1    ? \@get
      : $get[0];
}

my $_actual_key = sub {
    my ( $self, $key ) = @_;
    if ( ref $key ) {
        my $new_key = $self->{curr_key_of}{ refaddr $key};
        return refaddr $key unless defined $new_key;
        $key = $new_key;
    }
    return $key;
};
```

Single key fetch overhead!

```
sub get {
    my ( $self, @keys ) = @_;
    my @get;
    foreach my $key (@keys) {
        $key = $self->$_actual_key($key);
        next unless defined $key;
        my $exists = $self->exists($key);
        if ( $self->{is_strict} && !$exists ) {
            $self->$_croak("Cannot get non-existent key ($key)");
        }
        if ($exists) {
            CORE::push @get, $self->{array_for}[ $self->$_index($key) + 1 ];
        }
        elsif ( @keys > 1 ) {
            CORE::push @get, undef;
        }
        else {
            return;
        }
    }
    return wantarray ? @get
      : @keys > 1     ? \@get
      : $get[0];
}

my $_actual_key = sub {
    my ( $self, $key ) = @_;
    if ( ref $key ) {
        my $new_key = $self->{curr_key_of}{ refaddr $key};
        return refaddr $key unless defined $new_key;
        $key = $new_key;
    }
    return $key;
};
```

Tie::Hash::Indexed

XS, but flawed

- Opaque data: Perl hash of doubly-linked list of C structs
- Fails tests since Perl 5.18 randomization
- Actually, not all that fast (benchmarks later)

What else?

Special-purpose or weird

- Tie::Array::AsHash — array elements split with separator; tie API only
- Tie::Hash::Array — ordered alphabetically; tie API only
- Tie::InsertOrderHash — ordered by insertion; tie API only
- Tie::StoredOrderHash — ordered by last update; tie API only
- Array::Assign — arrays with named access; restricted keys
- Array::OrdHash — overloads array/hash deref and uses internal tied data
- Data::Pairs — array of key-value hashrefs; allows duplicate keys
- Data::OMap — array of key-value hashrefs; no duplicate keys
- Data::XHash — blessed, tied hashref with doubly-linked-list

Complexity → Bad

What is the simplest thing that could work?

```
bless { {a=>1, b=>2}, [‘a’, ‘b’] }
```

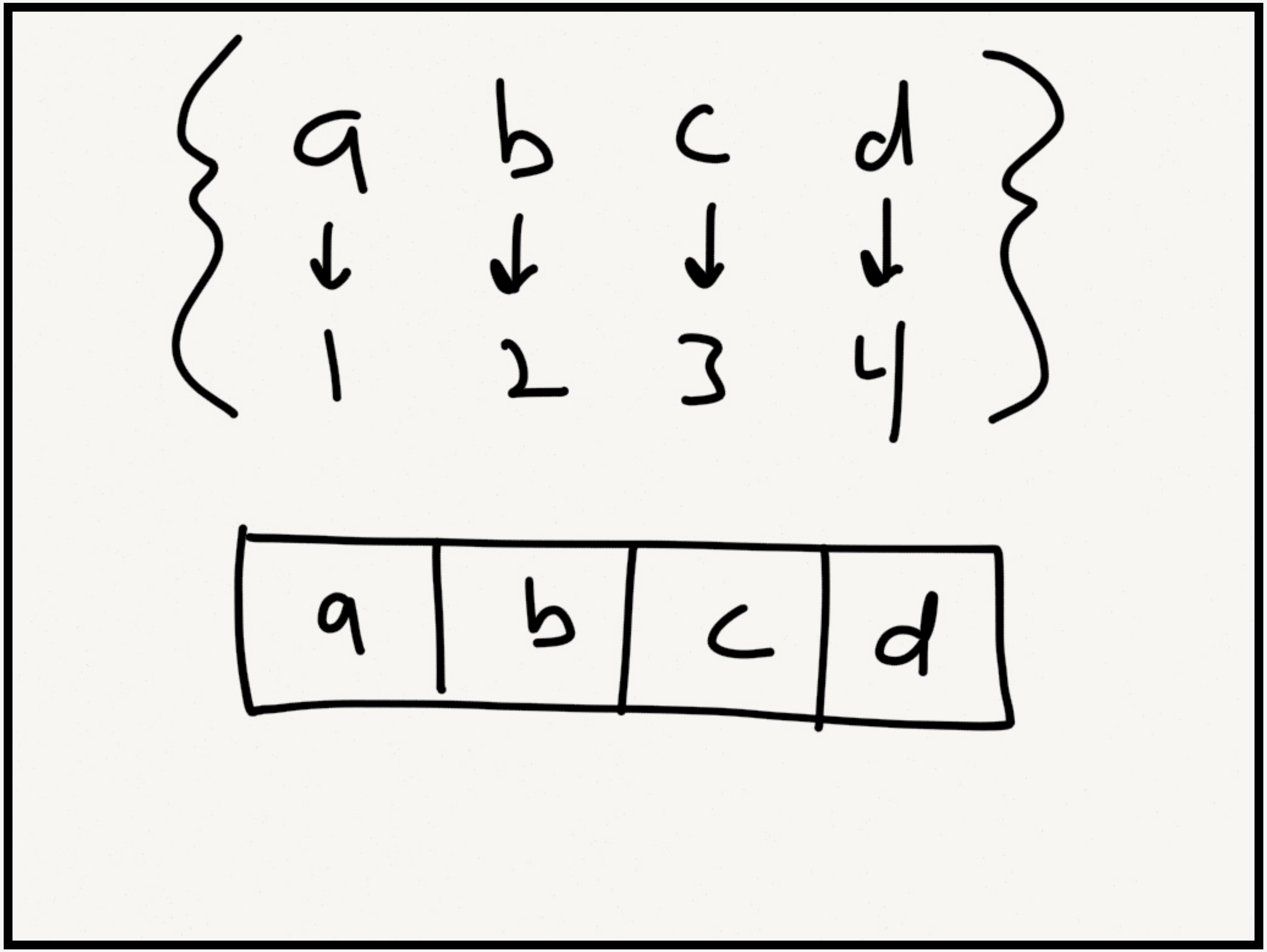
- Hash of keys and values
- Array of key order

I couldn’t find it on CPAN

So I wrote it

Hash::Ordered

```
Hash::Ordered->new(a=>1,b=>2,c=>3,d=>4);
```



Cheap get

```
sub get {
    my ( $self, $key ) = @_;
    return $self->[_DATA]{$key};
}
```

- only 2 dereferences
- no need to test exists()

Cheap-ish set

```
sub set {
    my ( $self, $key, $value ) = @_;
    if ( !exists $self->[_DATA]{$key} ) {
        push @{ $self->[_KEYS] }, $key;
    }
    return $self->[_DATA]{$key} = $value;
}
```

- exists plus 4-6 dereferences and maybe push
- comparable to Tie::IxHash::FETCH

Got my shallow copy

```
sub clone {
    my ( $self, @keys ) = @_;
    my $clone;
    if (@keys) {
        my %subhash;
        @subhash{@keys} = @{ $self->[_DATA] }{@keys};
        $clone = [ \%subhash, \@keys ];
    }
    else {
        $clone = [ { %{ $self->[_DATA] } }, [ @{ $self->[_KEYS] } ] ];
    }
    return bless $clone, ref $self;
}
```

Got my iterator

```perl
sub iterator {
    my ( $self, @keys ) = @_;
    @keys = @{ $self->[_KEYS] } unless @keys;
    my $data = $self->[_DATA];
    return sub {
        return unless @keys;
        my $key = CORE::shift(@keys);
        return ( $key => $data->{$key} );
    };
}
```

But, delete is expensive

```
sub delete {
    my ( $self, $key ) = @_;
    if ( exists $self->[_DATA]{$key} ) {
        my $r = $self->[_KEYS];
        my $i = List::Util::first { $r->[$_] eq $key } 0 .. $#$r;
        splice @$r, $i, 1;
        return delete $self->[_DATA]{$key};
    }
    return undef;
}
```

Good tradeoffs?

- It's ::Tiny — only about 130 SLOC
- Faster get and set
- Faster copy
- Slower delete

But is it actually fast?

Benchmarking is not profiling

Profiling → finding hot spots in code

Benchmarking → comparing different code to do the same thing

Scale can reveal 'Big-O' issues in algorithms

Constants matter
even for O(1)

Combinations

- Different ordered hash modules
- Different operations (create, get, set)
- Different scales (10, 100, 1000 elements)

Benchmarking tools

- Benchmark.pm
- Dumbbench
- Other stuff on CPAN

Don’t make timing distribution assumptions

Kolmogorov–Smirnov test

- Compare empirical CDFs
- Non-parametric
- Unequal-variance
- Sensitive to CDF location and shape

Doesn’t exist on CPAN

yet
I haven’t written it ^

KISS → Benchmark.pm

Benchmark.pm is verbose

```
Benchmark: running a, b, each for at least 5 CPU seconds...
         a: 10 wallclock secs ( 5.14 usr +  0.13 sys =  5.27 CPU) @ 3835055.60/s (n=20210743)
         b:  5 wallclock secs ( 5.41 usr +  0.00 sys =  5.41 CPU) @ 1574944.92/s (n=8520452)
       Rate    b    a
b 1574945/s   -- -59%
a 3835056/s 144%   --
```

- Big test matrix is unreadable
- Lots of detail I don’t care about

Approach

- Given a hash of test labels and code refs
- Output timings in descending order
- Repeat at different scales

```
use Benchmark qw( countit );

use constant COUNT => 5; # CPU seconds

sub time_them {
    my (%mark) = @_;
    my %results;

    for my $k ( sort keys %mark ) {
        my $res = countit( COUNT, $mark{$k} );
        my $iter_s = $res->iters / ( $res->cpu_a + 1e-9 );
        $results{$k} = $iter_s;
    }

    printf( "%20s %d/s\n", $_, $results{$_} )
      for  sort { $results{$b} <=> $results{$a} }
           keys %results;

    say "";
}
```

Use varied, but constant test data across runs

```perl
use Math::Random::MT::Auto qw/irand/;

use constant NUMS => [ 10, 100, 1000 ];

my %PAIRS = (
    map {
        $_ => [ map { irand() => irand() } 1 .. $_ ]
    } @{ NUMS() }
);
```

Example: hash creation

```
for my $size ( @{ NUMS() } ) {
    say my $title = "Results for ordered hash creation for $size elements";
    my %mark;
    $mark{"h:o"} = sub { my $h = Hash::Ordered->new( @{ $PAIRS{$size} } ) };
    $mark{"t:ix_oo"} = sub { my $h = Tie::IxHash->new( @{ $PAIRS{$size} } ) };
    $mark{"t:ix_th"} = sub { tie my %h, 'Tie::IxHash', @{ $PAIRS{$size} } };
    $mark{"t:llh"} = sub { tie my %h, 'Tie::LLHash', @{ $PAIRS{$size} } };
    # …
    time_them(%mark);
}
```

Includes variations

```
for my $size ( @{ NUMS() } ) {
    say my $title = "Results for ordered hash creation for $size elements";
    my %mark;
    $mark{"h:o"} = sub { my $h = Hash::Ordered->new( @{ $PAIRS{$size} } ) };
    $mark{"t:ix_oo"} = sub { my $h = Tie::IxHash->new( @{ $PAIRS{$size} } ) };
    $mark{"t:ix_th"} = sub { tie my %h, 'Tie::IxHash', @{ $PAIRS{$size} } };
    $mark{"t:llh"} = sub { tie my %h, 'Tie::LLHash', @{ $PAIRS{$size} } };
    # …
    time_them(%mark);
}
```

Example: fetch elements

```
for my $size ( @{ NUMS() } ) {

    say my $title = "Results for fetching ~10% of $size elements";

    my $oh     = Hash::Ordered->new( @{ $PAIRS{$size} } );
    my $tix_oo = Tie::IxHash->new( @{ $PAIRS{$size} } );
    tie my %tix_th, 'Tie::IxHash',        @{ $PAIRS{$size} };
    tie my %tllh,   'Tie::LLHash',        @{ $PAIRS{$size} };
    # …

    my ( %mark, $v );
    my @keys = keys %{ { @{ $PAIRS{$size} } } };

    my $n = int( .1 * scalar @keys ) || 1;
    my @lookup = map { $keys[ int( rand( scalar @keys ) ) ] } 1 .. $n;

    $mark{"h:o"}     = sub { $v = $oh->get($_)       for @lookup };
    $mark{"t:ix_oo"} = sub { $v = $tix_oo->FETCH($_) for @lookup };
    $mark{"t:ix_th"} = sub { $v = $tix_th{$_}        for @lookup };
    $mark{"t:llh"}   = sub { $v = $tllh{$_}          for @lookup };
    # …

    time_them(%mark);
}
```

Pre-generates hashes

```
for my $size ( @{ NUMS() } ) {

    say my $title = "Results for fetching ~10% of $size elements";

    my $oh     = Hash::Ordered->new( @{ $PAIRS{$size} } );
    my $tix_oo = Tie::IxHash->new( @{ $PAIRS{$size} } );
    tie my %tix_th, 'Tie::IxHash',        @{ $PAIRS{$size} };
    tie my %tllh,   'Tie::LLHash',        @{ $PAIRS{$size} };
    # …

    my ( %mark, $v );
    my @keys = keys %{ { @{ $PAIRS{$size} } } };

    my $n = int( .1 * scalar @keys ) || 1;
    my @lookup = map { $keys[ int( rand( scalar @keys ) ) ] } 1 .. $n;

    $mark{"h:o"}     = sub { $v = $oh->get($_)       for @lookup };
    $mark{"t:ix_oo"} = sub { $v = $tix_oo->FETCH($_) for @lookup };
    $mark{"t:ix_th"} = sub { $v = $tix_th{$_}        for @lookup };
    $mark{"t:llh"}   = sub { $v = $tllh{$_}          for @lookup };
    # …

    time_them(%mark);
}
```

Pre-generates test keys

```
for my $size ( @{ NUMS() } ) {

    say my $title = "Results for fetching ~10% of $size elements";

    my $oh     = Hash::Ordered->new( @{ $PAIRS{$size} } );
    my $tix_oo = Tie::IxHash->new( @{ $PAIRS{$size} } );
    tie my %tix_th, 'Tie::IxHash',        @{ $PAIRS{$size} };
    tie my %tllh,   'Tie::LLHash',        @{ $PAIRS{$size} };
    # …

    my ( %mark, $v );
    my @keys = keys %{ { @{ $PAIRS{$size} } } };

    my $n = int( .1 * scalar @keys ) || 1;
    my @lookup = map { $keys[ int( rand( scalar @keys ) ) ] } 1 .. $n;

    $mark{"h:o"}     = sub { $v = $oh->get($_)       for @lookup };
    $mark{"t:ix_oo"} = sub { $v = $tix_oo->FETCH($_) for @lookup };
    $mark{"t:ix_th"} = sub { $v = $tix_th{$_}        for @lookup };
    $mark{"t:llh"}   = sub { $v = $tllh{$_}          for @lookup };
    # …

    time_them(%mark);
}
```

Benchmark just the fetch

```
for my $size ( @{ NUMS() } ) {

    say my $title = "Results for fetching ~10% of $size elements";

    my $oh     = Hash::Ordered->new( @{ $PAIRS{$size} } );
    my $tix_oo = Tie::IxHash->new( @{ $PAIRS{$size} } );
    tie my %tix_th, 'Tie::IxHash',        @{ $PAIRS{$size} };
    tie my %tllh,   'Tie::LLHash',        @{ $PAIRS{$size} };
    # …

    my ( %mark, $v );
    my @keys = keys %{ { @{ $PAIRS{$size} } } };

    my $n = int( .1 * scalar @keys ) || 1;
    my @lookup = map { $keys[ int( rand( scalar @keys ) ) ] } 1 .. $n;

    $mark{"h:o"}     = sub { $v = $oh->get($_)       for @lookup };
    $mark{"t:ix_oo"} = sub { $v = $tix_oo->FETCH($_) for @lookup };
    $mark{"t:ix_th"} = sub { $v = $tix_th{$_}        for @lookup };
    $mark{"t:llh"}   = sub { $v = $tllh{$_}          for @lookup };
    # …

    time_them(%mark);
}
```

Example: deleting elements

```
for my $size ( @{ NUMS() } ) {

    say my $title = "Results for creating $size element hash then deleting ~10%";

    my ( %mark, $v );
    my @keys = keys %{ { @{ $PAIRS{$size} } } };

    my $n = int( .1 * scalar @keys ) || 1;
    my @lookup = map { $keys[ int( rand( scalar @keys ) ) ] } 1 .. $n;

    $mark{"h:o"} = sub {
        my $oh = Hash::Ordered->new( @{ $PAIRS{$size} } );
        $oh->delete($_) for @lookup;
    };

    $mark{"t:ix_oo"} = sub {
        my $tix_oo = Tie::IxHash->new( @{ $PAIRS{$size} } );
        $tix_oo->DELETE($_) for @lookup;
    };

    # …

    time_them(%mark);
}
```

But, we can’t isolate delete

```
for my $size ( @{ NUMS() } ) {

    say my $title = "Results for creating $size element hash then deleting ~10%";

    my ( %mark, $v );
    my @keys = keys %{ { @{ $PAIRS{$size} } } };

    my $n = int( .1 * scalar @keys ) || 1;
    my @lookup = map { $keys[ int( rand( scalar @keys ) ) ] } 1 .. $n;

    $mark{"h:o"} = sub {
        my $oh = Hash::Ordered->new( @{ $PAIRS{$size} } );
        $oh->delete($_) for @lookup;
    };

    $mark{"t:ix_oo"} = sub {
        my $tix_oo = Tie::IxHash->new( @{ $PAIRS{$size} } );
        $tix_oo->DELETE($_) for @lookup;
    };

    # …

    time_them(%mark);
}
```

Results…

https://www.flickr.com/photos/tarikb/111831472/

Don’t web-surf while benchmarking!

Modules & abbreviations

- Hash::Ordered → h:o [data hash + keys array]
- Array::AsHash → a:ah [data array + index hash]
- Tie::IxHash → t:ix [tie + hash + 2 x array]
- Tie::LLHash → t:llh [tie + hash + 2LL]
- Tie::Hash::Indexed → t:h:i [XS + tie + hash + 2LL]
- Array::OrdHash → a:oh [overloaded + private ties]
- Data::XHash → d:xh [tie + double linked list]

Creation

```
    10 elements           100 elements          1000 elements

  t:h:i 129713/s        t:h:i 15026/s        a:ah_rf 1410/s
a:ah_rf 104034/s      a:ah_rf 14304/s          t:h:i 1285/s
    h:o  94121/s          h:o 10931/s            h:o 1022/s
a:ah_cp  62539/s      a:ah_cp  7512/s        a:ah_cp  763/s
t:ix_th  60136/s      t:ix_oo  7368/s        t:ix_oo  703/s
t:ix_oo  59895/s      t:ix_th  7161/s        t:ix_th  697/s
   a:oh  49399/s         a:oh  6572/s           a:oh  694/s
  t:llh  32122/s        t:llh  3306/s          t:llh  290/s
d:xh_rf  13288/s      d:xh_ls  1498/s        d:xh_rf  147/s
d:xh_ls  13223/s      d:xh_rf  1491/s        d:xh_ls  146/s
```

Fetch 10% of elements

```
      10 elements            100 elements           1000 elements

     h:o 1417712/s          h:o 244800/s            h:o 24871/s
d:xh_oo 1231973/s      d:xh_oo 181520/s        d:xh_oo 19125/s
t:ix_oo 1120271/s      t:ix_oo 175981/s        t:ix_oo 17655/s
  t:h:i  792250/s        t:h:i 132963/s          t:h:i 13407/s
d:xh_rf  722683/s      d:xh_rf  93519/s        d:xh_rf  9590/s
t:ix_th  624603/s      t:ix_th  82154/s        t:ix_th  8455/s
   a:oh  553755/s         a:oh  68270/s           a:oh  6995/s
  t:llh  504533/s        t:llh  57013/s          t:llh  5781/s
   a:ah  246063/s         a:ah  28280/s           a:ah  2219/s
```

Set 10% of elements

```
       10 elements              100 elements             1000 elements

     h:o 1353795/s              h:o 197232/s               h:o 20364/s
d:xh_oo   952485/s            t:h:i 131238/s             t:h:i 13254/s
  t:h:i   943983/s          d:xh_oo 121692/s           d:xh_oo 12512/s
t:ix_oo   923874/s          t:ix_oo 114869/s           t:ix_oo 11542/s
  t:llh   600717/s            t:llh  71720/s             t:llh  7295/s
d:xh_rf   568693/s          d:xh_rf  67130/s           d:xh_rf  7004/s
   a:oh   547233/s             a:oh  63634/s              a:oh  6376/s
t:ix_th   519939/s          t:ix_th  59784/s           t:ix_th  6175/s
   a:ah   164170/s             a:ah  16843/s              a:ah  1635/s
```

Adding elements to empty

```
      10 elements        100 elements        1000 elements

     h:o 367588/s         h:o 66495/s           h:o 7217/s
   t:h:i 300357/s       t:h:i 57307/s         t:h:i 6244/s
 t:ix_oo 263158/s     t:ix_oo 49676/s       t:ix_oo 5671/s
 t:ix_th 214085/s     t:ix_th 38222/s          a:oh 4335/s
   t:llh 187981/s        a:oh 35476/s       t:ix_th 4313/s
    a:oh 141308/s       t:llh 27998/s       d:xh_oo 2977/s
    a:ah  96523/s     d:xh_oo 24371/s         t:llh 2899/s
 d:xh_oo  87498/s     d:xh_rf 22326/s       d:xh_rf 2683/s
 d:xh_rf  84316/s        a:ah 14114/s          a:ah 1466/s
```

Deleting* 10% of keys

```
         10 elements              100 elements            1000 elements

   t:h:i 139517/s            t:h:i 16745/s             t:h:i 1604/s
         h:o 95284/s            h:o 6924/s                t:llh 269/s
       a:ah 66495/s         t:ix_oo 4063/s                 a:oh 171/s
  t:ix_oo 52892/s              a:oh 3963/s              d:xh_rf 146/s
  t:ix_th 50254/s           t:ix_th 3590/s                  h:o 144/s
       a:oh 45609/s            a:ah 3014/s              d:xh_oo 130/s
      t:llh 28599/s           t:llh 2459/s               t:ix_oo 85/s
  d:xh_rf 13223/s           d:xh_oo 1449/s               t:ix_th 77/s
  d:xh_oo 13173/s           d:xh_rf 1434/s                  a:ah 36/s
```

Output hash as a list

```
     10 elements             100 elements           1000 elements

   a:ah 290725/s          a:ah 39222/s          a:ah 3703/s
    h:o 170187/s           h:o 18839/s           h:o 1877/s
t:ix_oo  92118/s       t:ix_oo  9525/s       t:ix_oo  961/s
  t:h:i  80408/s         t:h:i  7742/s         t:h:i  768/s
t:ix_th  48756/s          a:oh  5081/s          a:oh  508/s
  t:llh  38509/s       t:ix_th  5014/s       t:ix_th  505/s
   a:oh  36126/s          d:xh  4160/s          d:xh  413/s
   d:xh  35766/s         t:llh  3841/s         t:llh  385/s
```

Conclusions…

Tying sucks

Module choice matters a lot

- 7 CPAN modules tested
- 10x performance difference on some tasks
- Look inside modules before you use them!

Simplicity pays off

- Less indirection
- Less memory allocation
- Fewer ops per call

Hash::Ordered::XS
might really rock!

Questions?