

Ruby master - Bug #10984

Hash#contain? to check whether hash contains other hash

03/19/2015 02:00 PM - olivierlacan (Olivier Lacan)

Status: Closed	
Priority: Normal	
Assignee: akr (Akira Tanaka)	
Target version:	
ruby -v:	Backport:
Description	
<p>Comparing hashes seems like a common practice but there currently isn't a method to ask a hash instance whether it includes another hash instance.</p> <p>The most intuitive method to reach for would be Hash#include? but it is in fact an alias to Hash#has_key?</p> <p>What I'm looking for can be achieved with:</p> <pre>class Hash def contain?(other) self.merge(other) == self end end</pre> <p>Here's a simple demo of #contain? in use:</p> <pre>{ a: true, b: false }.contain?({ a: true}) # => true { a: true, b: false }.contain?({ b: false}) # => true { a: true, b: false }.contain?({ a: false}) # => false { a: true, b: false }.contain?({ c: true}) # => false</pre> <p>One important note is that this method is <i>not checking for nested hash matches</i>. This may need to be addressed when the parameters include a nested hash perhaps.</p> <p>Thanks to Terence Lee's help, nobu created a patch for this feature last year. I've only modified the name of the method from his original patch and attached it to this issue.</p>	

Associated revisions

Revision d68c3ecf - 11/10/2015 05:02 AM - nobu (Nobuyoshi Nakada)

hash.c: compare methods

- hash.c (rb_hash_{le,lt,ge,gt}): new methods, Hash#<=, Hash#<, Hash#>=, Hash#>, to test if all elements of a hash are also included in another hash, and vice versa. [ruby-core:68561] [Feature #10984]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@52518 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 52518 - 11/10/2015 05:02 AM - nobu (Nobuyoshi Nakada)

hash.c: compare methods

- hash.c (rb_hash_{le,lt,ge,gt}): new methods, Hash#<=, Hash#<, Hash#>=, Hash#>, to test if all elements of a hash are also included in another hash, and vice versa. [ruby-core:68561] [Feature #10984]

Revision 52518 - 11/10/2015 05:02 AM - nobu (Nobuyoshi Nakada)

hash.c: compare methods

- `hash.c (rb_hash_{le,lt,ge,gt})`: new methods, `Hash#<=`, `Hash#<`, `Hash#>=`, `Hash#>`, to test if all elements of a hash are also included in another hash, and vice versa. [ruby-core:68561] [Feature #10984]

Revision 52518 - 11/10/2015 05:02 AM - nobu (Nobuyoshi Nakada)

hash.c: compare methods

- `hash.c (rb_hash_{le,lt,ge,gt})`: new methods, `Hash#<=`, `Hash#<`, `Hash#>=`, `Hash#>`, to test if all elements of a hash are also included in another hash, and vice versa. [ruby-core:68561] [Feature #10984]

Revision 52518 - 11/10/2015 05:02 AM - nobu (Nobuyoshi Nakada)

hash.c: compare methods

- `hash.c (rb_hash_{le,lt,ge,gt})`: new methods, `Hash#<=`, `Hash#<`, `Hash#>=`, `Hash#>`, to test if all elements of a hash are also included in another hash, and vice versa. [ruby-core:68561] [Feature #10984]

Revision 52518 - 11/10/2015 05:02 AM - nobu (Nobuyoshi Nakada)

hash.c: compare methods

- `hash.c (rb_hash_{le,lt,ge,gt})`: new methods, `Hash#<=`, `Hash#<`, `Hash#>=`, `Hash#>`, to test if all elements of a hash are also included in another hash, and vice versa. [ruby-core:68561] [Feature #10984]

Revision 59a3fe58 - 11/14/2015 08:13 AM - nobu (Nobuyoshi Nakada)

hash.c: compare methods [ci skip]

- `hash.c (rb_hash_{le,lt,ge,gt})`: [DOC] for [Feature #10984]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@52570 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 52570 - 11/14/2015 08:13 AM - nobu (Nobuyoshi Nakada)

hash.c: compare methods [ci skip]

- `hash.c (rb_hash_{le,lt,ge,gt})`: [DOC] for [Feature #10984]

Revision 52570 - 11/14/2015 08:13 AM - nobu (Nobuyoshi Nakada)

hash.c: compare methods [ci skip]

- `hash.c (rb_hash_{le,lt,ge,gt})`: [DOC] for [Feature #10984]

Revision 52570 - 11/14/2015 08:13 AM - nobu (Nobuyoshi Nakada)

hash.c: compare methods [ci skip]

- `hash.c (rb_hash_{le,lt,ge,gt})`: [DOC] for [Feature #10984]

Revision 52570 - 11/14/2015 08:13 AM - nobu (Nobuyoshi Nakada)

hash.c: compare methods [ci skip]

- `hash.c (rb_hash_{le,lt,ge,gt})`: [DOC] for [Feature #10984]

Revision 52570 - 11/14/2015 08:13 AM - nobu (Nobuyoshi Nakada)

hash.c: compare methods [ci skip]

- `hash.c (rb_hash_{le,lt,ge,gt})`: [DOC] for [Feature #10984]

History

#1 - 03/19/2015 02:11 PM - sferik (Erik Michaels-Ober)

I agree that `#include?` is a more consistent and appropriate name for this method. I would prefer to see that change introduced in Ruby 3.0 than settle for this suboptimal change in Ruby 2.

#2 - 03/19/2015 02:20 PM - olivierlacan (Olivier Lacan)

Erik Michaels-Ober wrote:

I agree that `#include?` is a more consistent and appropriate name for this method. I would prefer to see that change introduced in Ruby 3.0 than settle for this suboptimal change in Ruby 2.

From the many conversations I've had with folks after [publishing the original proposal for this](#) it seemed that my expectations for Hash#include?'s behavior didn't match many (if not most) people's expectations so it seemed unlikely that such a breaking change would ever be favored.

I don't think a brand new method would be such a bad idea in 2.3 because it would allow for a smoother upgrade path. People could start using Hash#contain? when 2.3 ships and then the internal implementation of Hash#include? could be swapped in 3.0 if that is deemed a good idea later on.

#3 - 03/21/2015 11:45 PM - shevegen (Robert A. Heiler)

I have no particular pro or contra opinion. I merely wish to think that #include? is a better name because Array also has include? and String also has include?. While a Hash#include? may not be exactly the same as the other two #includes?, I think the name would be ok. #contain? feels a bit weird, I mean you could use Array#contain? too (on the other hand, you could use an alias just as .map vs. .collect; I myself use solely .map, other people can use .collect, things are fine) ok so bottom line from me, I think Hash#include? is a perfectly suitable name for it too as Erik wrote. Perhaps for Ruby 3.0

#4 - 08/18/2015 04:30 AM - olivierlacan (Olivier Lacan)

I didn't notice the notes from [DevelopersMeeting20150514Japan](#) before so I'll link to my original musings about Hash#include? which later evolved into this feature/patch: [Proposal for a better Ruby Hash#include?](#)

I'm aware this post needs revision and more concrete examples and use cases. I'll try to bring that together soon.

#5 - 11/03/2015 11:38 PM - olivierlacan (Olivier Lacan)

- Assignee set to akr (Akira Tanaka)

Responding to feedback from Akira Tanaka and Nobuyoshi Nakada at [DevelopersMeeting20150514Japan](#)

akr: "contain" is too general. "subhash"?

You mean something like this?

```
{ a: 1, b: 2 }.subhash?({ b: 2 })
```

Semantically, this feels strange to me. It doesn't seem obvious at all which hash we're checking for a subhash on and I would expect a lot of confusion with a method name like this. Compare to:

```
{ a: 1, b: 2 }.contains?({ b: 2 })
```

I believe contains is semantically far more self-evident.

It also seems odd to introduce a sub<class>? method name for this since I'm not aware of any similar method names for classes that would have similar behavior.

n0kada: "contain?" seems similar to "include?"

It is. Sadly, I've been told repeatedly that it's a bad idea to try to change the behavior of include?. I would prefer replacing the existing include? but I will settle for contains? for now because the meaning of "contain" focuses on what's inside the object under observation and is far more commonly used than "comprise":

```
contain |kənˈtān|  
verb [ with obj. ]
```

1. have or hold (someone or something) within: coffee cans that once contained a full pound of coffee.
- be made up of (a number of things); consist of: borscht can contain mainly beets or a number of vegetables
.
- (of a number) be divisible by (a factor) without a remainder.

akr: do we really use? we need concrete examples.

Yes, RSpec has an ad-hoc implementation of this feature in its include matcher:

https://github.com/rspec/rspec-expectations/blob/bb731e29f7800f5cef736cf8850293276a0d3f90/lib/rspec/matchers/built_in/include.rb#L94-L97

RSpec has been downloaded 29 Million times on RubyGems. I think this is a legitimate use case. This would simplify not only RSpec's internal code for Hash matchers, but any existing application who depends on this code, for a relatively minimal impact on the core Hash codebase (see provided patch).

I expanded on my original proposal (since then changed from Hash#include? to Hash#contains?) here:

<http://olivierlacan.com/posts/proposal-for-a-better-ruby-hash-include/>

#6 - 11/04/2015 03:48 AM - funny_falcon (Yura Sokolov)

What if

```
{b: 1} === {a: 2, b: 1}
```

then

```
h = {a: 2, b: 1}
case h
when {b: 1}
  puts "got it"
end
```

□□□□□□□□

#7 - 11/09/2015 08:56 AM - matz (Yukihiko Matsumoto)

Hash#contain? has slight ambiguity problem. I'd vote for adding >=, along with <=.

Matz.

#8 - 11/09/2015 08:57 AM - ko1 (Koichi Sasada)

Discussion: https://docs.google.com/document/d/1D0Eo5N7NE_unlySOKG9IVj_eyXf66BQPM4PKp7NvMyQ/pub

Feel free to continue discussion on this ticket.

#9 - 11/09/2015 03:06 PM - olivierlacan (Olivier Lacan)

Yukihiko Matsumoto wrote:

Hash#contain? has slight ambiguity problem. I'd vote for adding >=, along with <=.

Matz.

Thanks for considering this feature, Matz. :-)

If I understand correctly, the following examples would be correct?

```
{ a: 1, b: 2 } >= { b: 2 }
=> true
{ b: 2 } <= { a: 1, b: 2 }
=> true

# also
{ b: 2 } >= { b: 2 }
=> true
{ a: 1 } >= { b: 2 }
=> false

{ b: 2 } <= { b: 2 }
=> true
{ b: 2 } <= { a: 1 }
=> false
```

I think the versatility of this alone would make it even more useful than the one direction method I suggested.

#10 - 11/10/2015 03:30 AM - nobu (Nobuyoshi Nakada)

If we'll introduce Hash#<= and Hash#>=, then Hash#< and Hash#> too?

Hash will include Comparable with Hash#<=>?

#11 - 11/10/2015 03:45 AM - akr (Akira Tanaka)

Nobuyoshi Nakada wrote:

If we'll introduce Hash#<= and Hash#>=, then Hash#< and Hash#> too?

Maybe.

Usefulness of Hash#< and Hash#> is not discussed well, though.

Hash will include Comparable with Hash#<=>?

No. It is clearly stated by matz.

```
% ruby -e '
class Hash
  def <=(other)
    self.merge(other) == other
  end
  def >=(other)
    self.merge(other) == self
  end
  def <(other)
    self <= other && self != other
  end
  def >(other)
    self >= other && self != other
  end
end
hs = [{a:1, b:2}, {a:1, b:2, c:3}]
ops = %w[<= >= < >]
ops.each {|op|
  hs.each {|h1|
    hs.each {|h2|
      puts "#{h1} #{op} #{h2} = #{h1.send(op, h2)}"
    }
  }
}
'
{:a=>1, :b=>2} <= {:a=>1, :b=>2} = true
{:a=>1, :b=>2} <= {:a=>1, :b=>2, :c=>3} = true
{:a=>1, :b=>2, :c=>3} <= {:a=>1, :b=>2} = false
{:a=>1, :b=>2, :c=>3} <= {:a=>1, :b=>2, :c=>3} = true
{:a=>1, :b=>2} >= {:a=>1, :b=>2} = true
{:a=>1, :b=>2} >= {:a=>1, :b=>2, :c=>3} = false
{:a=>1, :b=>2, :c=>3} >= {:a=>1, :b=>2} = true
{:a=>1, :b=>2, :c=>3} >= {:a=>1, :b=>2, :c=>3} = true
{:a=>1, :b=>2} < {:a=>1, :b=>2} = false
{:a=>1, :b=>2} < {:a=>1, :b=>2, :c=>3} = true
{:a=>1, :b=>2, :c=>3} < {:a=>1, :b=>2} = false
{:a=>1, :b=>2, :c=>3} < {:a=>1, :b=>2, :c=>3} = false
{:a=>1, :b=>2} > {:a=>1, :b=>2} = false
{:a=>1, :b=>2} > {:a=>1, :b=>2, :c=>3} = false
{:a=>1, :b=>2, :c=>3} > {:a=>1, :b=>2} = true
{:a=>1, :b=>2, :c=>3} > {:a=>1, :b=>2, :c=>3} = false
```

#12 - 11/10/2015 05:02 AM - nobu (Nobuyoshi Nakada)

- Status changed from Open to Closed

Applied in changeset r52518.

hash.c: compare methods

- hash.c (rb_hash_{le,lt,ge,gt}): new methods, Hash#<=, Hash#<, Hash#>=, Hash#>, to test if all elements of a hash are also included in another hash, and vice versa. [ruby-core:68561] [Feature [#10984](#)]

#13 - 11/11/2015 05:33 AM - akr (Akira Tanaka)

Akira Tanaka wrote:

```
% ruby -e '
class Hash
  def <=(other)
    self.merge(other) == other
  end
  def >=(other)
    self.merge(other) == self
  end
  def <(other)
    self <= other && self != other
  end
  def >(other)
```

```

    self >= other && self != other
  end
end
hs = [{a:1, b:2}, {a:1, b:2, c:3}]
ops = %w[<= >= < >]
ops.each {|op|
  hs.each {|h1|
    hs.each {|h2|
      puts "#{h1} #{op} #{h2} = #{h1.send(op, h2)}"
    }
  }
}
',
{:a=>1, :b=>2} <= {:a=>1, :b=>2} = true
{:a=>1, :b=>2} <= {:a=>1, :b=>2, :c=>3} = true
{:a=>1, :b=>2, :c=>3} <= {:a=>1, :b=>2} = false
{:a=>1, :b=>2, :c=>3} <= {:a=>1, :b=>2, :c=>3} = true
{:a=>1, :b=>2} >= {:a=>1, :b=>2} = true
{:a=>1, :b=>2} >= {:a=>1, :b=>2, :c=>3} = false
{:a=>1, :b=>2, :c=>3} >= {:a=>1, :b=>2} = true
{:a=>1, :b=>2, :c=>3} >= {:a=>1, :b=>2, :c=>3} = true
{:a=>1, :b=>2} < {:a=>1, :b=>2} = false
{:a=>1, :b=>2} < {:a=>1, :b=>2, :c=>3} = true
{:a=>1, :b=>2, :c=>3} < {:a=>1, :b=>2} = false
{:a=>1, :b=>2, :c=>3} < {:a=>1, :b=>2, :c=>3} = false
{:a=>1, :b=>2} > {:a=>1, :b=>2} = false
{:a=>1, :b=>2} > {:a=>1, :b=>2, :c=>3} = false
{:a=>1, :b=>2, :c=>3} > {:a=>1, :b=>2} = true
{:a=>1, :b=>2, :c=>3} > {:a=>1, :b=>2, :c=>3} = false

```

For the record, this sample implementation was wrong.

It should consider that two hashes may have different values for same key.

```

% ruby -e '
class Hash
  def <=(other)
    self.merge(other) {|k,v1,v2| v1 } == other
  end
  def >=(other)
    self.merge(other) {|k,v1,v2| v2 } == self
  end
  def <(other)
    self <= other && self != other
  end
  def >(other)
    self >= other && self != other
  end
end
hs = [{a:1}, {a:2}]
ops = %w[<= >= < >]
ops.each {|op|
  hs.each {|h1|
    hs.each {|h2|
      puts "#{h1} #{op} #{h2} = #{h1.send(op, h2)}"
    }
  }
}
',
{:a=>1} <= {:a=>1} = true
{:a=>1} <= {:a=>2} = false
{:a=>2} <= {:a=>1} = false
{:a=>2} <= {:a=>2} = true
{:a=>1} >= {:a=>1} = true
{:a=>1} >= {:a=>2} = false
{:a=>2} >= {:a=>1} = false
{:a=>2} >= {:a=>2} = true
{:a=>1} < {:a=>1} = false
{:a=>1} < {:a=>2} = false
{:a=>2} < {:a=>1} = false
{:a=>2} < {:a=>2} = false
{:a=>1} > {:a=>1} = false
{:a=>1} > {:a=>2} = false
{:a=>2} > {:a=>1} = false

```

```
{:a=>2} > {:a=>2} = false
```

Note that nobu's implementation already committed has no problem.

#14 - 11/17/2015 11:13 PM - prijutme4ty (Ilya Vorontsov)

Hello everyone.

I urge to remove Hash comparison methods and to stick to methods like #contain. Or at least to return nil instead of false for comparison of non-comparable hashes. Underlying reasons are strictly mathematical but have far-reaching consequences.

Usually we deal with linearly ordered sets or totally ordered (like usual numbers or string are) i.e. such sets that either $a \leq b$ or $b \leq a$ for every two elements a and b of a set.

Comparison can be generalized for posets or partially ordered sets. They don't require that any two elements are comparable. Set of hashes is a typical example of a partially ordered set (see "Partial ordered set" or "Hasse diagram" in wikipedia).

One must not implement $a \leq b$ for unrelated elements because if such comparison returns any certain result either true or false - then its negation would be counterintuitive. I'm not a proponent of current ruby approach of `Class#<=>` because ordinary intuition based on everyday use of totally ordered sets suggest that this code would be correct which is definitely false:

```
if String <= Fixnum
  puts 'String is a Fixnum subclass'
else
  puts 'Fixnum is a String subclass'
end
```

But at least `String <=> Fixnum` is neither true or false but nil which allow us to distinguish such situations. nil result is properly handled by Comparable methods like #sort. Thus `[String, Fixnum].sort` will raise.

So why one can sort this array and which result does one expect?:

```
[{}, {a:1,b:2}, {c:3}, {a:1}, {b:2}].sort
```

That's why, I insist, comparison of non-comparable hashes at least must return nil. As a more strict approach one can raise exception when try to compare hashes but it makes the main use-case impractical. But I can't see why one want to deal with such a controversial methods when #contain and #included_by will be enough for this not-so-often task.

As an example of why implementing #<=> for posets is not a good idea, lets consider this typical hand-written qsort implementation.

```
def qsort(arr)
  return arr if arr.size <= 1
  pivot = arr[arr.length / 2]
  left = arr.select{|el| el < pivot }
  right = arr.select{|el| el > pivot }
  central = arr.select{|el| el == pivot }
  qsort(left) + central + qsort(right)
end
```

Okay. Now lets run and see how this "obvious" algorithm loses values.

```
qsort( [ {}, {a:1,b:2}, {c:3}, {a:1}, {b:2} ] )
# => [ {}, {c=>3} ]
```

Surely, sorting is already implemented, but this problem persist in every place where one suggest that $a < b$, $a == b$ and $a > b$ are the only possible alternatives - thus in almost every if-else pair.

I ask a community think one more time about consequences of such a decision.

Ilya

#15 - 11/17/2015 11:54 PM - olivierlacan (Olivier Lacan)

Ilya Vorontsov wrote:

So why one can sort this array and which result does one expect?:

```
[{}, {a:1,b:2}, {c:3}, {a:1}, {b:2}].sort
```

This is the actual result with 2.3.0-preview1:

```
RUBY_VERSION
=> "2.3.0"
[ {}, {a:1,b:2}, {c:3}, {a:1}, {b:2} ].sort
ArgumentError: comparison of Hash with Hash failed
```

Maybe I'm reading you too quickly or I'm too tired but did you note that `<=>` is **not** implemented on Hash? Matz made that clear and Akira Tanaka clarified it in [this response](#).

#16 - 11/18/2015 12:59 AM - prijutme4ty (Ilya Vorontsov)

I've missed absence of `<=>` first. Yes, you are right. And it can slightly reduce damage.

But anyway it doesn't resolve issues with misinterpretation of comparison negation. My `qsort` function is just the one problem which lie on surface. There are lots of `(a < b) ...; else ...` constructions in lots of codebases. I'm sure that such behaviour will lead to tons of subtle bugs.

Also It's highly probable that some libraries will be subjected to malicious inputs which will cause infinite recursion bombs or other threats. Here I've slightly changed `qsort` code using `#reject` instead of `#select`. Not a very natural code but it's still correct:

```
def qsort(arr)
  return arr if arr.size <= 1
  pivot = arr[arr.length / 2]
  left = arr.reject{|el| el >= pivot }
  right = arr.reject{|el| el <= pivot }
  central = arr.select{|el| el == pivot }
  qsort(left) + central + qsort(right)
end
```

This code will not reduce size as in my previous example but contrariwise will expand array size and break asymptotical estimations of CPU time.

```
qsort( [ {}, {a:1,b:2}, {c:3}, {a:1}, {b:2} ] )
# => [ {}, {:b=>2}, {:a=>1}, {:b=>2}, {:a=>1, :b=>2}, {:c=>3}, {:b=>2}, {:a=>1}, {:b=>2}, {:a=>1, :b=>2} ]
```

Look at expansion rate.

```
qsort( 10.times.map{|i| {i => i} } ).size # => 1023
qsort( 20.times.map{|i| {i => i} } ).size # => 1048575
qsort( 25.times.map{|i| {i => i} } ).size # => 33554431
```

The last array of 25 single-element hashes is sorted in a minute or so. It can easily hang server.

Ok, I specially wrote code subjected to this attack. But in a large codebase of ruby community there would be places which have similar problems (which are not problems for totally ordered sets like numbers). It can be hard to find and exploit it, but if you did - you have broken the server. Just imagine that some code which earlier just raised an exception due to bad input now ruins your application.

#17 - 11/19/2015 01:38 AM - prijutme4ty (Ilya Vorontsov)

- *Tracker changed from Feature to Bug*

Akira Tanaka wrote:

```
% ruby -e '
class Hash
  def <=(other)
    self.merge(other) == other
  end
  def >=(other)
    self.merge(other) == self
  end
  def <(other)
    self <= other && self != other
  end
  def >(other)
    self >= other && self != other
  end
end
end
```

This implies that

```
{a: 1, b: 2} <= {a: 1, b: 3}
{a: 1, b: 3} <= {a: 1, b: 2}
```

Is that expected? This breaks the main use-case in testing method results.

#18 - 11/19/2015 08:57 AM - nobu (Nobuyoshi Nakada)

It's an easy code to show the concept.
Both return false.

#19 - 11/20/2015 03:34 PM - cvss (Kirill Vechera)

I second Ilya's opinion regarding partially ordered sets. But propose to implement the comparison similar to classes - Hash and Class both satisfy reflexive, antisymmetric, and transitive relations. So like the Class, Hash can implement $\lt\Rightarrow$, returning nil on noncomparable argument.

Take a look on classes:

```
class A; end
class B; end
class BB < B; end
```

How they are comparing:

```
> A < B
=> nil
> B < BB
=> false
> BB < B
=> true
> A <=> B
=> nil
> B <=> BB
=> 1
```

So the similar approach could be implemented for hashes, with inversed signification for the \lt and \gt , to realize this behavior:

```
> {a: 1} < {b: 2}
=> nil
> {a: 1} < {a: 1, b: 2}
=> true
> {a: 1; b: 2} < {a: 1}
=> false
> {a: 1} <=> {b: 2}
=> nil
> {a: 1} <=> {a: 1, b: 2}
=> -1
```

Regarding qsort - partially ordered sets imply topological sorting, not kind of sorting algorithms for strictly ordered sets. So qsort should not work on array of hashes.

Files

Hash#contain_patch	2.22 KB	03/19/2015	olivierlacan (Olivier Lacan)
--------------------	---------	------------	------------------------------