

NAME

Array::IntSpan – Handles arrays of scalars or objects using integer ranges

SYNOPSIS

```
use Array::IntSpan;

my $foo = Array::IntSpan->new([0, 59, 'F'], [60, 69, 'D'], [80, 89, 'B']);

print "A score of 84% results in a ".$foo->lookup(84)."\n";
unless (defined($foo->lookup(70))) {
    print "The grade for the score 70% is currently undefined.\n";
}

$foo->set_range(70, 79, 'C');
print "A score of 75% now results in a ".$foo->lookup(75)."\n";

$foo->set_range(0, 59, undef);
unless (defined($foo->lookup(40))) {
    print "The grade for the score 40% is now undefined.\n";
}

$foo->set_range(87, 89, 'B+');
$foo->set_range(85, 100, 'A');
$foo->set_range(100, 1_000_000, 'A+');
```

DESCRIPTION

Array::IntSpan brings the speed advantages of Set::IntSpan (written by Steven McDougall) to arrays. Uses include manipulating grades, routing tables, or any other situation where you have mutually exclusive ranges of integers that map to given values.

The new version of Array::IntSpan is also able to consolidate the ranges by comparing the adjacent values of the range. If 2 adjacent values are identical, the 2 adjacent ranges are merged.

Ranges of objects

Array::IntSpan can also handle objects instead of scalar values.

But for the consolidation to work, the payload class must overload the "", eq and == operators to perform the consolidation comparisons.

When a get_range method is called to a range of objects, it will return a new range of object references. These object references points to the objects stored in the original range. In other words the objects contained in the returned range are **not** copied.

Thus if the user calls a methods on the objects contained in the returned range, the method is actually invoked on the objects stored in the original range.

When a get_range method is called on a range of objects, several things may happen:

- The get_range spans empty slots. By default the returned range will skip the empty slots. But the user may provide a callback to create new objects (for instance). See details below.
- The get_range splits existing ranges. By default, the split range will contains the same object reference. The user may provide callback to perform the object copy so that the split range will contains different objects. See details below.

Ranges specified with integer fields

- Array::IntSpan::IP is also provided with the distribution. It lets you use IP addresses in any of three forms (dotted decimal, network string, and integer) for the indices into the array. See the POD for that module for more information. See Array::IntSpan::IP for details.
- Array::IntSpan::Fields is also provided with the distribution. It let you specify an arbitrary specification to handle ranges with strings made of several integer separated by dots (like IP addresses

of ANSI SS7 point codes). See Array::IntSpan::Fields for details.

METHODS

new (...)

The new method takes an optional list of array elements. The elements should be in the form [start_index, end_index, value]. They should be in sorted order and there should be no overlaps. The internal method `_check_structure` will be called to verify the data is correct. If you wish to avoid the performance penalties of checking the structure, you can use `Data::Dumper` to dump an object and use that code to reconstitute it.

clear

Clear the range.

set_range (start, end, value [, code ref])

This method takes three parameters – the `start_index`, the `end_index`, and the `value`. If you wish to erase a range, specify `undef` for the `value`. It properly deals with overlapping ranges and will replace existing data as appropriate. If the new range lies after the last existing range, the method will execute in $O(1)$ time. If the new range lies within the existing ranges, the method executes in $O(n)$ time, where n is the number of ranges. It does not consolidate contiguous ranges that have the same `value`.

If you have a large number of inserts to do, it would be beneficial to sort them first. Sorting is $O(n \lg(n))$, and since appending is $O(1)$, that will be considerably faster than the $O(n^2)$ time for inserting n unsorted elements.

The method returns 0 if there were no overlapping ranges and 1 if there were.

The optional `code ref` is called back when an existing range is split. For instance if the original range is [0,10,\$foo_obj] and `set_range` is called with [5,7,\$bar_obj'], the callback will be called twice:

```
$callback->(0, 4, $foo_obj)
$callback->(8, 10, $foo_obj)
```

It will be the callback responsibility to make sure that the range 0-4 and 7-10 holds 2 *different* objects.

set(index, value [, code ref])

Set a single value. This may split an existing range. Actually calls:

```
set_range( index, index, value [, code ref] )
```

set_range_as_string (index, string [, code ref])

Set one or several ranges specified with a string. Ranges are separated by “-”. Several ranges can be specified with commas.

Example:

```
set_range_as_string( '1-10,13, 14-20', 'foo'
```

White space are ignored.

get_range (start, end [, filler | undef , copy_cb [, set_cb]])

This method returns a range (actually an Array::IntSpan object) from `start` to `end`.

If `start` and `end` span empty slot in the original range, `get_range` will skip the empty slots. If a `filler` value is provided, `get_range` will fill the slots with it.

```
original range      : [2-4, X], [7-9, Y], [12-14, Z]
get_range(3, 8)     : [3-4, X], [7-8, Y]
get_range(2, 10, f) : [3-4, X], [5-6, f], [7-8, Y]
```

If the `filler` parameter is a CODE reference, the filler value will be the one returned by the sub ref. The sub ref is invoked with (`start`, `end`), i.e. the range of the empty span to fill (`get_range(5, 6)` in the example above). When handling object, the sub ref can invoke an object constructor.

If `start` or `end` split an original range in 2, the default behavior is to copy the value or object ref contained in the original range:

```

original range      : [1-4, X]
split range        : [1-1, X], [2-2, X], [3-4, X]
get_range(2)       : [2-2, X]

```

If the original range contains object, this may lead to disappointing results. In the example below the 2 ranges contains references (`obj_a`) that points to the same object:

```

original range      : [1-4, obj_a]
split range        : [1-1, obj_a], [2-2, obj_a], [3-4, obj_a]
get_range(2)       : [2-2, obj_a]

```

Which means that invoking a method on the object returned by `get_range(2)` will also be invoked on the range 1-4 of the original range which may not be what you want.

If `get_range` is invoked with a copy parameter (actually a code reference), the result of this routine will be stored in the split range *outside* of the `get_range`:

```

original range      : [1-4, X]
get_range(2)       : [2-2, X]
split range        : [1-1, copy_of_X], [2-2, X], [3-4, copy_of_X]

```

When dealing with object, the sub ref should provide a copy of the object:

```

original range      : [1-4, obj_a]
get_range(2)       : [2-2, obj_a]
split range        : [1-1, obj_a1], [2-2, obj_a], [3-4, obj_a2]

```

Note that the `obj_a` contained in the `split range` and the `obj_a` contained in the returned range point to the *same object*.

The sub ref is invoked with `(start, end, obj_a)` and is expected to return a copy of `obj_a` that will be stored in the split ranges. In the example above, 2 different copies are made: `obj_a1` and `obj_a2`.

Last, a 3rd callback may be defined by the user: the `set_cb`. This callback will be used when the range start or end that holds an object changes. In the example above, the `set_cb` will be called this way:

```
$obj_a->&$set_cb(2, 2) ;
```

As a matter of fact, the 3 callback can be used in the same call. In the example below, `get_range` is invoked with 3 subs refs: `&f`, `&cp`, `&set`:

```

original range      : [1-4, obj_a], [7-9, obj_b]
get_range(3-8, ...) : [3-4, obj_a], [5-6, obj_fill], [7-8, obj_b]
split range        : [1-2, obj_a1], [3-4, obj_a], [5-6, obj_fill],
                   [7-8, obj_b], [9-9, obj_b1]

```

To obtain this, `get_range` will perform the following calls:

```

$obj_fill = &f ;
$obj_a1 = &cp(5, 6, obj_a) ;
&set(3, 4, $obj_a) ;
$obj_b = &cp(9, 9, obj_b) ;
&set(7-8, obj_b) ;

```

get_range_list

In scalar context, returns a list of range in a string like: "1-5, 7, 9-11".

In list context returns a list of list, E.g. ([1, 5], [7, 7], 9, 11).

lookup(index)

This method takes as a single parameter the `index` to look up. If there is an appropriate range, the method will return the associated value. Otherwise, it returns `undef`.

get_element(element_number)

Returns an array containing the Nth range element:

```
( start, end, value )
```

consolidate([bottom, top , [set_cb]])

This function scan the range from the range index `bottom` to `top` and compare the values held by the adjacent ranges. If the values are identical, the adjacent ranges are merged.

The comparison is made with the `==` operator. Objects stored in the range **must** overload the `==` operator. If not, the comparison will be made with the standard stringification of an object and the merge will never happen.

If provided, the `set_cb` will be invoked on the contained object after 2 ranges are merged.

For instance, if the `"$obj_a"` eq `"$obj_b"`:

```
original range      : [1-4, obj_a], [5-9, obj_b]
consolidate(0, 1, \&set) : [1-9, obj_a]
```

And `consolidate` will perform this call:

```
&$set(1, 9, obj_a) ;
```

Consolidate the whole range when called without parameters.

AUTHOR

- Toby Everett, teverett@alacom.att.com
- Dominique Dumont, ddumont@cpan.org

Copyright (c) 2000 Toby Everett. Copyright (c) 2003–2004,2014 Dominique Dumont. All rights reserved. This program is free software.

This module is distributed under the Artistic License. See http://www.ActiveState.com/corporate/artistic_license.htm or the license that comes with your perl distribution.