Data Structures

Version 5.1

February 14, 2011

```
(require data)
```

This manual documents data structure libraries available in the `data` collection.

1 Imperative Queues

```
(require data/queue)
```

This module provides a simple mutable queue representation, first-in/first-out only. Operations on queues mutate it in a thread-unsafe way.

```
(make-queue) → queue/c
```

Produces an empty queue.

```
(enqueue! q v) → void?
  q : queue/c
  v : any/c
```

Adds an element to the back of a queue.

```
(dequeue! q) → any/c
  q : nonempty-queue/c
```

Removes an element from the front of a nonempty queue, and returns that element.

Examples:

```
(define q (make-queue))
> (enqueue! q 1)
> (dequeue! q)
1
> (enqueue! q 2)
> (enqueue! q 3)
> (dequeue! q)
2
> (dequeue! q)
3
```

```
(queue->list queue) → (listof any/c)
  queue : queue/c
```

Returns an immutable list containing the elements of the queue in the order the elements were added.

Examples:

```
(define queue (make-queue))
> (enqueue! queue 8)
```

```
> (enqueue! queue 9)
> (enqueue! queue 0)
> (queue->list queue)
'(8 9 0)
```

```
(queue-length queue) → integer?
  queue : queue/c
```

Returns the number of elements in the queue.

Examples:

```
(define queue (make-queue))
> (queue-length queue)
0
> (enqueue! queue 5)
> (enqueue! queue 12)
> (queue-length queue)
2
> (dequeue! queue)
5
> (queue-length queue)
1
```

```
(queue-empty? q) → boolean?
  q : queue/c
```

Recognizes whether a queue is empty or not.

Examples:

```
(define q (make-queue))
> (queue-empty? q)
#t
> (enqueue! q 1)
> (queue-empty? q)
#f
> (dequeue! q)
1
> (queue-empty? q)
#t
```

```
(queue? v) → boolean?
  v : any/c
```

This predicate recognizes queues.

Examples:

```
> (queue? (make-queue))
#t
> (queue? 'not-a-queue)
#f
```

```
(in-queue queue) → sequence?
  queue : queue?
```

Returns a sequence whose elements are the elements of *queue*.

```
queue/c : flat-contract?
nonempty-queue/c : flat-contract?
```

These contracts recognize queues; the latter requires the queue to contain at least one value.

2 Growable Vectors

```
(require data/gvector)
```

A growable vector (gvector) is a mutable sequence whose length can change over time. A gvector also acts as a dictionary (`dict?` from `racket/dict`), where the keys are zero-based indexes and the values are the elements of the gvector. A gvector can be extended by adding an element to the end, and it can be shrunk by removing any element, although removal can take time linear in the number of elements in the gvector.

Two gvectors are `equal?` if they contain the same number of elements and if the contain equal elements at each index.

```
(make-gvector [#:capacity capacity]) → gvector?
  capacity : exact-positive-integer? = 10
```

Creates a new empty gvector with an initial capacity of *capacity*.

```
(gvector elem ...) → gvector?
  elem : any/c
```

Creates a new gvector containing each *elem* in order.

```
(gvector? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a gvector, `#f` otherwise.

```
(gvector-ref gv index [default]) → any/c
  gv : gvector?
  index : exact-nonnegative-integer?
  default : any/c = (error ....)
```

Returns the element at index *index*, if *index* is less than `(gvector-count gv)`. Otherwise, *default* is invoked if it is a procedure, returned otherwise.

```
(gvector-add! gv value ...) → void?
  gv : gvector?
  value : any/c
```

Adds each *value* to the end of the gvector *gv*.

```
(gvector-set! gv index value) → void?
  gv : gvector?
  index : (and/c exact-nonnegative-integer?
                 (</c (+ 1 (gvector-count gv))))
  value : any/c
```

Sets the value at index *index* to be *value*. If *index* is `(gvector-count gv)`—that is, one more than the greatest used index—the effect is the same as `(gvector-add! gv value)`.

```
(gvector-remove! gv index) → void?
  gv : gvector?
  index : (and/c exact-nonnegative-integer?
                 (</c (gvector-count gv)))
```

Removes the item at *index*, shifting items at higher indexes down. Takes time proportional to `(- (gvector-count gv) index)`.

```
(gvector-count gv) → exact-nonnegative-integer?
  gv : gvector?
```

Returns the number of items in *gv*.

```
(gvector->vector gv) → vector?
  gv : gvector?
```

Returns a vector of length `(gvector-count gv)` containing the elements of *gv* in order.

```
(in-gvector gv) → sequence?
  gv : gvector?
```

Returns a sequence whose elements are the elements of *gv*. Mutation of *gv* while the sequence is running changes the elements produced by the sequence. To obtain a sequence from a snapshot of *gv*, use `(in-vector (gvector->vector gv))` instead.

```
(for/gvector (for-clause ...) body ...+)
(for*/gvector (for-clause ...) body ...+)
```

Analogous to `for/list` and `for*/list`, but constructs a gvector instead of a list.

Unlike `for/list`, the *body* may return zero or multiple values; all returned values are added to the gvector, in order, on each iteration.

3 Orders and Ordered Dictionaries

```
(require data/order)
```

This library defines *orders* and the *ordered dictionary* generic interface.

```
ordering/c : flat-contract?
```

Contract for orderings, represented by the symbols `'=`, `'<`, and `'>`.

```
prop:ordered-dict : (struct-type-property/c
                     (vector-immutableof e/c e/c s/c s/c s/c s/c))
```

Struct-type property for defining new ordered dictionary types. The value associated with `prop:ordered-dict` should be an immutable vector of six procedures, two "extrema" procedures and four "search" procedures. The extrema procedures must satisfy *e/c* and the search procedures must satisfy *s/c*:

```
e/c = (->i ([d ordered-dict?])
           [_ (d) (or/c #f (dict-iter-contract d))])
s/c = (->i ([d ordered-dict?]
            [k (d) (dict-key-contract d)])
            [_ (d) (or/c #f (dict-iter-contract d))])
```

The procedures are implementations of the following generic functions:

- `dict-iterate-least`
- `dict-iterate-greatest`
- `dict-iterate-least/>?`
- `dict-iterate-least/>=?`
- `dict-iterate-greatest/<?`
- `dict-iterate-greatest/<=?`

A struct type that implements `prop:ordered-dict` must also implement `prop:dict`.

```
(ordered-dict? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is an instance of a struct implementing the ordered dictionary interface (via `prop:ordered-dict`).

```
(dict-iterate-least dict) → any/c
  dict : ordered-dict?
(dict-iterate-greatest dict) → any/c
  dict : ordered-dict?
```

Returns the position of the least (greatest) key in the ordered dictionary *dict*. If *dict* is empty, `#f` is returned.

```
(dict-iterate-least/>? dict key) → any/c
  dict : ordered-dict?
  key : any/c
(dict-iterate-least/>=? dict key) → any/c
  dict : ordered-dict?
  key : any/c
(dict-iterate-greatest/<? dict key) → any/c
  dict : ordered-dict?
  key : any/c
(dict-iterate-greatest/<=? dict key) → any/c
  dict : ordered-dict?
  key : any/c
```

Returns the position of the least key greater than *key*, the least key greater than or equal to *key*, the greatest key less than *key*, and the greatest key less than or equal to *key*, respectively. If no key satisfies the criterion, `#f` is returned.

```
(order name domain-contract comparator)
 → (and/c order? procedure?)
  name : symbol?
  domain-contract : contract?
  comparator : (-> any/c any/c ordering/c)
(order name domain-contract =? <? [>?]) → (and/c order? procedure?)
  name : symbol?
  domain-contract : contract?
  =? : (-> any/c any/c boolean?)
  <? : (-> any/c any/c boolean?)
  >? : (-> any/c any/c boolean?) = (lambda (x y) (<? y x))
```

Produces a named order object encapsulating a domain contract and a comparator function. If a single procedure is given, it is used directly as the comparator. If two or three procedures are given, they are used to construct the comparator.

The *domain-contract* is not applied to the comparison function; rather, clients of the order are advised to incorporate the domain contracts into their own contracts. For example, when a splay-tree (see `data/splay-tree`) is constructed with an order, it applies the domain-contract to its keys. Thus the contract is checked once per dictionary procedure call, rather than on every comparison.

An order object is applicable as a procedure; it behaves as its comparator.

Examples:

```
> (define string-order (order 'string-order string? string=? string<?))
> (string-order "abc" "acdc")
'<
> (string-order "x" 12)
string=?: expects type <string> as 2nd argument, given: 12;
other arguments were: "x"
```

`(order? x) → boolean?`
  `x : any/c`

Returns `#t` if *x* is an order object, `#f` otherwise.

`(order-comparator ord) → (-> any/c any/c ordering/c)`
  `ord : order?`

Extracts the comparator function from an order object.

`(order-domain-contract ord) → contract?`
  `ord : order?`

Extracts the domain contract from an order object.

`(order-=? ord) → (-> any/c any/c boolean?)`
  `ord : order?`
`(order-<? ord) → (-> any/c any/c boolean?)`
  `ord : order?`

Returns a procedure representing the order's equality relation or less-than relation, respectively.

`real-order : order?`

The order of the real numbers. The domain of `real-order` excludes `+nan.0` but includes `+inf.0` and `-inf.0`. The standard numeric comparisons (`=`, `<`) are used; exact `1` is equal to inexact `1.0`.

Examples:

```
> (real-order 1.0 1)
'=
> (real-order 5 7)
'<
> (real-order 9.0 3.4)
'>
> (real-order 1 +inf.0)
'<
> (real-order 5 -inf.0)
'>
```

`datum-order : order?`

An ad hoc order that encompasses many built-in Racket data types. The `datum-order` comparator orders values of the same data type according to the data type's natural order: `string=?`, `string<?` for strings, for example (but see the warning about numbers below). Different data types are ordered arbitrarily but contiguously; for example, all strings sort before all vectors, or vice versa. Programs should not rely on the ordering of different data types.

The order is designed so that lists, vectors, and prefab structs are ordered lexicographically.

Warning! The `datum-order` is not compatible with the standard numeric order; all exact numbers are ordered before all inexact numbers. This allows `1` to be considered distinct from `1.0`, for example.

The following built-in data types are currently supported: numbers, strings, bytes, keywords, symbols, booleans, characters, null, pairs, vectors, boxes, and prefab structs.

Examples:

```
> (datum-order 1 2)
'<
> (datum-order 8 5.0)
'<
> (datum-order 3+5i 3+2i)
'>
> (datum-order '(a #:b c) '(a #:c d c))
'<
> (datum-order "apricot" "apple")
'>
> (datum-order '#(1 2 3) '#(1 2))
'>
> (datum-order '#(1 2 3) '#(1 3))
'<
> (datum-order 'apple (box "candy"))
```

’<

4 Splay Trees

```
(require data/splay-tree)
```

Splay trees are an efficient data structure for mutable dictionaries with totally ordered keys. They were described in the paper “Self-Adjusting Binary Search Trees” by Daniel Sleator and Robert Tarjan in Journal of the ACM 32(3) pp652-686.

A splay-tree is a ordered dictionary (`dict?` and `ordered-dict?`).

```
(make-splay-tree [ord
                  #:key-contract key-contract
                  #:value-contract value-contract]) → splay-tree?
  ord : order? = datum-order
  key-contract : contract? = any/c
  value-contract : contract? = any/c
```

Makes a new empty splay-tree. The splay tree uses *ord* to order keys; in addition, the domain contract of *ord* is combined with *key-contract* to check keys.

Examples:

```
> (define splay-tree
    (make-splay-tree (order 'string-order string? string=? string<?)))
> (splay-tree-set! splay-tree "dot" 10)
> (splay-tree-set! splay-tree "cherry" 500)
> (dict-map splay-tree list)
'(("cherry" 500) ("dot" 10))
> (splay-tree-ref splay-tree "dot")
10
> (splay-tree-remove! splay-tree "cherry")
> (splay-tree-count splay-tree)
1
> (splay-tree-set! splay-tree 'pear 3)
contract violation: expected <string?>, given: 'pear
  contract on splay-tree-set! from
    (file
     /var/tmp/racket/collects/data/splay-tree.rkt)
  blaming top-level
  contract:
    (->i
     ((s splay-tree?) (key (s) ...) (v (s) ...))
     (_r void?))
  at: <collects>/data/splay-tree.rkt:1112.2
```

```
(make-adjustable-splay-tree [#:key-contract key-contract
                             #:value-contract value-contract])
 → splay-tree?
  key-contract : contract? = any/c
  value-contract : contract? = any/c
```

Makes a new empty splay-tree that permits only exact integers as keys (in addition to any constraints imposed by *key-contract*). The resulting splay tree answers true to `adjustable-splay-tree?` and supports efficient key adjustment.

Examples:

```
> (define splay-tree (make-adjustable-splay-tree))
> (splay-tree-set! splay-tree 3 'apple)
> (splay-tree-set! splay-tree 6 'cherry)
> (dict-map splay-tree list)
'((3 apple) (6 cherry))
> (splay-tree-ref splay-tree 3)
'apple
> (splay-tree-remove! splay-tree 6)
> (splay-tree-count splay-tree)
1
```

```
(splay-tree? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a splay-tree, `#f` otherwise.

```
(adjustable-splay-tree? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a splay-tree that supports key adjustment; see `splay-tree-contract!` and `splay-tree-expand!`.

```
(splay-tree-ref s key [default]) → any
  s : splay-tree?
  key : any/c
  default : any/c = (lambda () (error ....))
(splay-tree-set! s key value) → void?
  s : splay-tree?
  key : any/c
  value : any/c
```

```
(splay-tree-remove! s key) → void?
  s : splay-tree?
  key : any/c
(splay-tree-count s) → exact-nonnegative-integer?
  s : splay-tree?
(splay-tree-iterate-first s) → (or/c #f splay-tree-iter?)
  s : splay-tree?
(splay-tree-iterate-next s iter) → (or/c #f splay-tree-iter?)
  s : splay-tree?
  iter : splay-tree-iter?
(splay-tree-iterate-key s iter) → any/c
  s : splay-tree?
  iter : splay-tree-iter?
(splay-tree-iterate-value s iter) → any/c
  s : splay-tree?
  iter : splay-tree-iter?
```

Implementations of `dict-ref`, `dict-set!`, `dict-remove!`, `dict-count`, `dict-iterate-first`, `dict-iterate-next`, `dict-iterate-key`, and `dict-iterate-value`, respectively.

```
(splay-tree-remove-range! s from to) → void?
  s : splay-tree?
  from : any/c
  to : any/c
```

Removes all keys in [*from*, *to*); that is, all keys greater than or equal to *from* and less than *to*.

This operation takes *O(N)* time, or *O(log N)* time if `(adjustable-splay-tree? s)`.

```
(splay-tree-contract! s from to) → void?
  s : adjustable-splay-tree?
  from : exact-integer?
  to : exact-integer?
```

Like `splay-tree-remove-range!`, but also decreases the value of all keys greater than or equal to *to* by `(- to from)`.

This operation is only allowed on adjustable splay trees, and it takes *O(log N)* time.

```
(splay-tree-expand! s from to) → void?
  s : adjustable-splay-tree?
  from : exact-integer?
```

```
  to : exact-integer?
```

Increases the value of all keys greater than or equal to *from* by `(- to from)`.

This operation is only allowed on adjustable splay trees, and it takes *O(log N)* time.

```
(splay-tree-iterate-least s) → (or/c #f splay-tree-iter?)
  s : splay-tree
(splay-tree-iterate-greatest s) → (or/c #f splay-tree-iter?)
  s : splay-tree
(splay-tree-iterate-least/>? s key) → (or/c #f splay-tree-iter?)
  s : splay-tree?
  key : any/c
(splay-tree-iterate-least/>=? s key)
 → (or/c #f splay-tree-iter?)
  s : splay-tree?
  key : any/c
(splay-tree-iterate-greatest/<? s key)
 → (or/c #f splay-tree-iter?)
  s : splay-tree?
  key : any/c
(splay-tree-iterate-greatest/<=? s key)
 → (or/c #f splay-tree-iter?)
  s : splay-tree?
  key : any/c
```

Implementations of `dict-iterate-least`, `dict-iterate-greatest`, `dict-iterate-least/>?`, `dict-iterate-least/>=?`, `dict-iterate-greatest/<?`, and `dict-iterate-greatest/<=?`, respectively.

```
(splay-tree-iter? x) → boolean?
  x : any/c
```

Returns `#t` if *x* represents a position in a splay-tree, `#f` otherwise.

```
(splay-tree->list s) → (listof pair?)
  s : splay-tree?
```

Returns an association list with the keys and values of *s*, in order.

5 Skip Lists

```
(require data/skip-list)
```

Skip lists are a simple, efficient data structure for mutable dictionaries with totally ordered keys. They were described in the paper “Skip Lists: A Probabilistic Alternative to Balanced Trees” by William Pugh in Communications of the ACM, June 1990, 33(6) pp668-676.

A skip-list is an ordered dictionary (`dict?` and `ordered-dict?`). It also supports extensions of the dictionary interface for iterator-based search and mutation.

```
(make-skip-list [ord
                 #:key-contract key-contract
                 #:value-contract value-contract]) → skip-list?
  ord : order? = datum-order
  key-contract : contract? = any/c
  value-contract : contract? = any/c
```

Makes a new empty skip-list. The skip-list uses *ord* to order keys; in addition, the domain contract of *ord* is combined with *key-contract* to check keys.

Examples:

```
> (define skip-list (make-skip-list real-order))
> (skip-list-set! skip-list 3 'apple)
> (skip-list-set! skip-list 6 'cherry)
> (dict-map skip-list list)
'((3 apple) (6 cherry))
> (skip-list-ref skip-list 3)
'apple
> (skip-list-remove! skip-list 6)
> (skip-list-count skip-list)
1
```

```
(make-adjustable-skip-list [#:key-contract key-contract
                            #:value-contract value-contract])
 → adjustable-skip-list?
  key-contract : contract? = any/c
  value-contract : contract? = any/c
```

Makes a new empty skip-list that permits only exact integers as keys (in addition to any constraints imposed by *key-contract*). The resulting skip-list answers true to `adjustable-skip-list?` and supports key adjustment.

```
(skip-list? v) → boolean?
```

```
v : any/c
```

Returns `#t` if *v* is a skip-list, `#f` otherwise.

```
(adjustable-skip-list? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a skip-list that supports key adjustment; see `skip-list-contract!` and `skip-list-expand!`.

```
(skip-list-ref skip-list key [default]) → any/c
  skip-list : skip-list?
  key : any/c
  default : any/c = (lambda () (error ....))
(skip-list-set! skip-list key value) → void?
  skip-list : skip-list?
  key : any/c
  value : any/c
(skip-list-remove! skip-list key) → void?
  skip-list : skip-list?
  key : any/c
(skip-list-count skip-list) → exact-nonnegative-integer?
  skip-list : skip-list?
(skip-list-iterate-first skip-list) → (or/c skip-list-iter? #f)
  skip-list : skip-list?
(skip-list-iterate-next skip-list iter)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  iter : skip-list-iter?
(skip-list-iterate-key skip-list iter) → any/c
  skip-list : skip-list?
  iter : skip-list-iter?
(skip-list-iterate-value skip-list iter) → any/c
  skip-list : skip-list?
  iter : skip-list-iter?
```

Implementations of `dict-ref`, `dict-set!`, `dict-remove!`, `dict-count`, `dict-iterate-first`, `dict-iterate-next`, `dict-iterate-key`, and `dict-iterate-value`, respectively.

```
(skip-list-remove-range! skip-list from to) → void?
  skip-list : skip-list?
  from : any/c
  to : any/c
```

Removes all keys in [*from*, *to*); that is, all keys greater than or equal to *from* and less than *to*.

```
(skip-list-contract! skip-list from to) → void?
  skip-list : adjustable-skip-list?
  from : exact-integer?
  to : exact-integer?
```

Like `skip-list-remove-range!`, but also decreases the value of all keys greater than or equal to *to* by `(- to from)`.

This operation takes time proportional to the number of elements with keys greater than or equal to *to*.

```
(skip-list-expand! skip-list from to) → void?
  skip-list : adjustable-skip-list?
  from : exact-integer?
  to : exact-integer?
```

Increases the value of all keys greater than or equal to *from* by `(- to from)`.

This operation takes time proportional to the number of elements with keys greater than or equal to *from*.

```
(skip-list-iterate-least/>? skip-list key)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  key : any/c
(skip-list-iterate-least/>=? skip-list key)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  key : any/c
(skip-list-iterate-greatest/<? skip-list
                                key)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  key : any/c
(skip-list-iterate-greatest/<=? skip-list
                                 key)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  key : any/c
```

Implementations of `dict-iterate-least`, `dict-iterate-greatest`, `dict-iterate-`

least/>?, dict-iterate-least/>=?, dict-iterate-greatest/<?, and dict-iterate-greatest/<=?, respectively.

```
(skip-list-iter? v) → boolean?
  v : any/c
```

Returns #t if *v* represents a position in a skip-list, #f otherwise.

```
(skip-list->list skip-list) → (listof pair?)
  skip-list : skip-list?
```

Returns an association list with the keys and values of *skip-list*, in order.

6 Interval Maps

```
(require data/interval-map)
```

An interval-map is a mutable data structure that maps *half-open* intervals of exact integers to values. An interval-map is queried at a discrete point, and the result of the query is the value mapped to the interval containing the point.

Internally, interval-maps use a splay-tree (`data/splay-tree`) of intervals for efficient query and update, including efficient contraction and expansion of intervals.

Interval-maps implement the dictionary (`racket/dict`) interface to a limited extent. Only `dict-ref` and the iteration-based methods (`dict-iterate-first`, `dict-map`, etc) are supported. For the iteration-based methods, the mapping's keys are considered the pairs of the start and end positions of the mapping's intervals.

Examples:

```
> (define r (make-interval-map))
> (interval-map-set! r 1 5 'apple)
> (interval-map-set! r 6 10 'pear)
> (interval-map-set! r 3 7 'banana)
> (dict-map r list)
'(((1 . 3) apple) ((3 . 7) banana) ((7 . 10) pear))
```

```
(make-interval-map [#:key-contract key-contract
                    #:value-contract value-contract])
 → interval-map?
  key-contract : contract? = any/c
  value-contract : contract? = any/c
```

Makes a new empty interval-map.

```
(interval-map? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is an interval-map, `#f` otherwise.

```
(interval-map-ref interval-map
                  position
                 [default])    → any/c
  interval-map : interval-map?
  position : exact-integer?
  default : any/c = (lambda () (error ....))
```

Return the value associated with *position* in *interval-map*. If no mapping is found, *default* is applied if it is a procedure, or returned otherwise.

```
(interval-map-set! interval-map
                   start
                   end
                   value)       → void?
  interval-map : interval-map?
  start : exact-integer?
  end : exact-integer?
  value : any/c
```

Updates *interval-map*, associating every position in [*start*, *end*) with *value*.

Existing interval mappings contained in [*start*, *end*) are destroyed, and partly overlapping intervals are truncated. See `interval-map-update*!` for an updating procedure that preserves distinctions within [*start*, *end*).

```
(interval-map-update*! interval-map
                       start
                       end
                       updater
                      [default])     → void?
  interval-map : interval-map?
  start : exact-integer?
  end : exact-integer?
  updater : (-> any/c any/c)
  default : any/c = (lambda () (error ....))
```

Updates *interval-map*, associating every position in [*start*, *end*) with the result of applying *updater* to the position's previously associated value, or to the default value produced by *default* if no mapping exists.

Unlike `interval-map-set!`, `interval-map-update*!` preserves existing distinctions within [*start*, *end*).

```
(interval-map-remove! interval-map
                      start
                      end)         → void?
  interval-map : interval-map?
  start : (or/c exact-integer? -inf.0)
  end : (or/c exact-integer? +inf.0)
```

Removes the value associated with every position in [*start*, *end*).

```
(interval-map-contract! interval-map
                        start
                        end)          → void?
  interval-map : interval-map?
  start : exact-integer?
  end : exact-integer?
```

Contracts `interval-map`'s domain by removing all mappings on the interval [`start`, `end`) and decreasing intervals initally after `end` by `(- end start)`.

If `start` is not less than `end`, an exception is raised.

```
(interval-map-expand! interval-map
                      start
                      end)          → void?
  interval-map : interval-map?
  start : exact-integer?
  end : exact-integer?
```

Expands `interval-map`'s domain by introducing a gap [`start`, `end`) and increasing intervals initially after `start` by `(- end start)`.

If `start` is not less than `end`, an exception is raised.

```
(interval-map-cons*! interval-map
                     start
                     end
                     v
                    [default])      → void?
  interval-map : interval-map?
  start : any/c
  end : any/c
  v : any/c
  default : any/c = null
```

Same as the following:

```
(interval-map-update*! interval-map start end
                       (lambda (old) (cons v old))
                       default)
```

```
(interval-map-iter? v) → boolean?
  v : any/c
```

Returns `#t` if *v* represents a position in an interval-map, `#f` otherwise.

7 Binary Heaps

```
(require data/heap)
```

Binary heaps are a simple implementation of priority queues.

```
(make-heap <=?) → heap?
  <=? : (-> any/c any/c any/c)
```

Makes a new empty heap using `<=?` to order elements.

```
(heap? x) → boolean?
  x : any/c
```

Returns `#t` if `x` is a heap, `#f` otherwise.

```
(heap-count h) → exact-nonnegative-integer?
  h : heap?
```

Returns the number of elements in the heap.

```
(heap-add! h v ...) → void?
  h : heap?
  v : any/c
```

Adds each `v` to the heap.

```
(heap-add-all! h v) → void?
  h : heap?
  v : (or/c list? vector? heap?)
```

Adds each element contained in `v` to the heap, leaving `v` unchanged.

```
(heap-min h) → any/c
  h : heap?
```

Returns the least element in the heap `h`, according to the heap's ordering. If the heap is empty, an exception is raised.

```
(heap-remove-min! h) → void?
  h : heap?
```

Removes the least element in the heap *h*. If the heap is empty, an exception is raised.

```
(vector->heap <=? items) → heap?
  <=? : (-> any/c any/c any/c)
  items : vector?
```

Builds a heap with the elements from *items*. The vector is not modified.

```
(heap->vector h) → vector?
  h : heap?
```

Returns a vector containing the elements of heap *h* in the heap's order. The heap is not modified.

```
(heap-copy h) → heap?
  h : heap?
```

Makes a copy of heap *h*.

```
(heap-sort! <=? v) → void?
  <=? : (-> any/c any/c any/c)
  v : vector?
```

Sorts vector *v* using the comparison function *<=?*.