Data: Data Structures

Version 5.3.5

June 18, 2013

```
(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, providing first-in/first-out semantics.

Operations on queues mutate it in a thread-unsafe way.

```
(make-queue) → queue?
```

Produces an empty queue.

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

Adds an element to the back of a queue.

This takes constant time, independent of the number of elements in `q`.

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

Adds an element to the front of a queue.

This takes constant time, independent of the number of elements in `q`.

```
(dequeue! q) → any/c
  q : non-empty-queue?
```

Removes an element from the front of a non-empty queue, and returns that element.

This takes constant time, independent of the number of elements in `q`.

Examples:

```
(define q (make-queue))

> (enqueue! q 1)

> (dequeue! q)
1
> (enqueue! q 2)
```

```
> (enqueue! q 3)

> (dequeue! q)
2
> (dequeue! q)
3
> (enqueue! q 2)

> (enqueue! q 1)

> (enqueue-front! q 3)

> (enqueue-front! q 4)

> (queue->list q)
'(4 3 2 1)
```

```
(queue-filter! q pred?) → void?
  q : queue?
  pred? : (-> any/c any/c)
```

Applies *pred?* to each element of the queue, removing any where *pred?* returns `#f`.

This takes time proportional to the number of elements in *q* (assuming that *pred?* takes constant time, independent of the number of elements in *q*). It does not allocate and it calls *pred?* exactly once for each element of *q*.

Examples:

```
(define q (make-queue))

> (enqueue! q 1)

> (enqueue! q 2)

> (enqueue! q 3)

> (enqueue! q 4)

> (queue-filter! q even?)

> (queue->list q)
'(2 4)
```

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

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

This takes time proportional to the number of elements in `q`.

Examples:

```
(define q (make-queue))

> (enqueue! q 8)

> (enqueue! q 9)

> (enqueue! q 0)

> (queue->list q)
'(8 9 0)
```

```
(queue-length queue) → exact-nonnegative-integer?
  queue : queue?
```

Returns the number of elements in the queue.

This takes constant time, independent of the number of elements in `q`.

Examples:

```
(define queue (make-queue))

> (queue-length q)
3
> (enqueue! q 5)

> (enqueue! q 12)

> (queue-length q)
5
> (dequeue! q)
8
> (queue-length q)
4
```

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

Recognizes whether a queue is empty or not.

This takes constant time, independent of the number of elements in `q`.

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.

This takes constant time, independent of the size of the argument `v`.

Examples:

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

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

This predicate recognizes non-empty queues.

This takes constant time, independent of the size of the argument `v`.

Examples:

```
> (non-empty-queue? (let ([q (make-queue)])
                      (enqueue! q 1)
                      q))
#t
> (non-empty-queue? (make-queue))
#f
> (non-empty-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 are provided for backwards compatibility. They are identical to queue? and non-empty-queue?, respectively.

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.

Operations on gvectors are not thread-safe.

```
(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-remove-last! gv) → any/c
  gv : gvector?
```

Removes the element at the end and returns it. Takes constant time.

```
(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.

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

Returns a list 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 '>.

```
gen:ordered-dict : any/c
```

A generic interface for defining new ordered dictionary types. Methods can be attached to the gen:ordered-dict interface using the #:methods keyword in a structure type definition. Two "extrema" methods and four "search" methods should be implemented. The extrema methods must satisfy *e/c* and the search methods 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 methods 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 gen:ordered-dict must also implement gen:dict.

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

A deprecated structure type property used to defined custom ordered dictionaries. Use gen:ordered-dict instead. Accepts a vector of 6 procedures with the same arguments as the methods of gen:ordered-dict.

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

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

```
(dict-iterate-least dict) → (or/c (dict-iter-contract dict) #f)
  dict : ordered-dict?
(dict-iterate-greatest dict)
 → (or/c (dict-iter-contract dict) #f)
  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)
 → (or/c (dict-iter-contract dict) #f)
  dict : ordered-dict?
  key : any/c
(dict-iterate-least/>=? dict key)
 → (or/c (dict-iter-contract dict) #f)
  dict : ordered-dict?
  key : any/c
(dict-iterate-greatest/<? dict key)
 → (or/c (dict-iter-contract dict) #f)
  dict : ordered-dict?
  key : any/c
(dict-iterate-greatest/<=? dict key)
 → (or/c (dict-iter-contract dict) #f)
  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=?: contract violation
  expected: string?
  given: 12
  argument position: 2nd
  other arguments...:
   "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 as well as prefab structs and fully-transparent structs. 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. Prefab and fully-transparent structs are ordered according to their most specific struct type, and prefab structs are ordered first by their prefab struct keys. The ordering of struct types is independent of the struct type hierarchy; a struct type may sort before one of its subtypes but after another.

Programs should not rely on the ordering of different data types, since it may change in future versions of Racket to improve comparison performance. The ordering of non-prefab struct types may change between one execution of a program and the next.

The order is guaranteed, however, to lexicographically sort proper lists, vectors, prefab structs, and fully-transparent structs. Improper lists sort lexicographically considered as

pairs, but the ordering of an improper list and its proper prefix, such as `'(a b . c)` and `'(a b)`, is not specified.

The `datum-order` comparator does not perform cycle-detection; comparisons involving cyclic data may diverge.

Warning: `datum-order` is not compatible with the standard numeric order; all exact numbers are ordered separately from all inexact numbers. Thus `1` is considered distinct from `1.0`, for example.

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

The following example comparisons are specified to return the results shown:

```
> (datum-order 1 2)
'<
> (datum-order 8.0 5.0)
'>
> (datum-order 'apple 'candy)
'<
> (datum-order '(a #:b c) '(a #:c d c))
'<
> (datum-order '(5 . 4) '(3 2 1))
'>
> (datum-order '(a b . c) '(a b . z))
'<
> (datum-order "apricot" "apple")
'>
> (datum-order '#(1 2 3) '#(1 2))
'>
> (datum-order '#(1 2 3) '#(1 3))
'<
> (datum-order (box 'car) (box 'candy))
'>
> (datum-order '#s(point a 1) '#s(point b 0))
'<
> (datum-order '#s(A 1 2) '#s(Z 3 4 5))
'<
> (struct fish (name) #:transparent)

> (datum-order (fish 'alewife) (fish 'sockeye))
'<
```

The following example comparisons are unspecified but consistent within all executions of

a single version of Racket:

```
(datum-order 1 2.0)
(datum-order 3+5i 3+2i)
(datum-order 'apple "zucchini")
(datum-order '(a b) '(a b . c))
(datum-order 0 'zero)
```

The following example comparison is unspecified but consistent within a single execution of a program:

```
(struct fowl (name) #:transparent)
(datum-order (fish 'alewife) (fowl 'dodo))
```

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?`).

Operations on splay-trees are not thread-safe. If a key in a splay-tree is mutated, the splay-tree's internal invariants may be violated, causing its behavior to become unpredictable.

```
(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)
splay-tree-set!: contract violation
  expected: string?
  given: 'pear
  in: the key argument of
      (->i
```

((s splay-tree?) (key (s) ...) (v (s) ...))
(_r void?))
contract from:
<collects>/data/splay-tree.rkt
blaming: top-level
at: <collects>/data/splay-tree.rkt:1115.2

```
(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.

Operations on skip-lists are not thread-safe. If a key in a skip-list is mutated, the skip-list's internal invariants may be violated, causings its behavior to become unpredictable.

```
(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
```

```
(skip-list-iterate-least skip-list) → (or/c skip-list-iter? #f)
  skip-list : skip-list?
(skip-list-iterate-greatest skip-list)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
```

Implementations of dict-iterate-least, dict-iterate-greatest, dict-iterate-least/>?, dict-iterate-least/>=?, dict-iterate-greatest/<?, dict-iterate-greatest/<=?, dict-iterate-least, 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))
```

Operations on interval-maps are not thread-safe.

```
(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-iterate-first interval-map)
 → (or/c interval-map-iter? #f)
  interval-map : interval-map?
```

```
(interval-map-iterate-next interval-map
                           iter)
 → (or/c interval-map-iter? #f)
  interval-map : interval-map?
  iter : interval-map-iter?
(interval-map-iterate-key interval-map
                          iter)        → pair?
  interval-map : interval-map?
  iter : interval-map-iter?
(interval-map-iterate-value interval-map
                            iter)        → any
  interval-map : interval-map?
  iter : interval-map-iter?
```

Implementations of `dict-iterate-first`, `dict-iterate-next`, `dict-iterate-key`, and `dict-iterate-value`, respectively.

```
(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.

Operations on binary heaps are not thread-safe.

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

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

Examples:

```
> (define a-heap-of-strings (make-heap string<=?))

> a-heap-of-strings
#<heap>
; With structs:
> (struct node (name val))

> (define (node<=? x y)
    (<= (node-val x) (node-val y)))

> (define a-heap-of-nodes (make-heap node<=?))

> a-heap-of-nodes
#<heap>
```

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

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

Examples:

```
> (heap? (make-heap <=))
#t
> (heap? "I am not a heap")
#f
```

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

Returns the number of elements in the heap.

Examples:

```
> (define a-heap (make-heap <=))

> (heap-add-all! a-heap '(7 3 9 1 13 21 15 31))

> (heap-count a-heap)
8
```

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

Adds each *v* to the heap.

Examples:

```
> (define a-heap (make-heap <=))

> (heap-add! a-heap 2009 1009)
```

```
(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.

Examples:

```
> (define heap-1 (make-heap <=))

> (define heap-2 (make-heap <=))

> (define heap-12 (make-heap <=))

> (heap-add-all! heap-1 '(3 1 4 1 5 9 2 6))

> (heap-add-all! heap-2 #(2 7 1 8 2 8 1 8))

> (heap-add-all! heap-12 heap-1)

> (heap-add-all! heap-12 heap-2)

> (heap-count heap-12)
16
```

```
(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.

Examples:

```
> (define a-heap (make-heap string<=?))

> (heap-add! a-heap "sneezy" "sleepy" "dopey" "doc"
             "happy" "bashful" "grumpy")

> (heap-min a-heap)
"bashful"
; Taking the min of the empty heap is an error:
> (heap-min (make-heap <=))
heap-min: empty heap
```

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

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

Examples:

```
> (define a-heap (make-heap string<=?))

> (heap-add! a-heap "fili" "fili" "oin" "gloin" "thorin"
                    "dwalin" "balin" "bifur" "bofur"
                    "bombur" "dori" "nori" "ori")

> (heap-min a-heap)
"balin"
> (heap-remove-min! a-heap)

> (heap-min a-heap)
"bifur"
```

```
(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.

Examples:

```
> (struct item (val frequency))

> (define (item<=? x y)
    (<= (item-frequency x) (item-frequency y)))

> (define some-sample-items
    (vector (item #\a 17) (item #\b 12) (item #\c 19)))

> (define a-heap (vector->heap item<=? some-sample-items))
```

(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.

Examples:

```
> (define word-heap (make-heap string<=?))

> (heap-add! word-heap "pile" "mound" "agglomerate" "cumulation")

> (heap->vector word-heap)
'#("agglomerate" "cumulation" "mound" "pile")
```

(heap-copy *h*) → heap?
  h : heap?

Makes a copy of heap *h*.

Examples:

```
> (define word-heap (make-heap string<=?))

> (heap-add! word-heap "pile" "mound" "agglomerate" "cumulation")

> (define a-copy (heap-copy word-heap))

> (heap-remove-min! a-copy)

> (heap-count word-heap)
4
> (heap-count a-copy)
3
```

(in-heap/consume! *heap*) → sequence?
  heap : heap?

Returns a sequence equivalent to *heap*, maintaining the heap's ordering. The heap is consumed in the process. Equivalent to repeated calling heap-min, then heap-remove-min!.

Examples:

```
> (define h (make-heap <=))

> (heap-add-all! h '(50 40 10 20 30))

> (for ([x (in-heap/consume! h)])
    (displayln x))
10
20
30
40
50

> (heap-count h)
0
```

(in-heap *heap*) → sequence?
  heap : heap?

Returns a sequence equivalent to *heap*, maintaining the heap's ordering. Equivalent to in-heap/consume! except the heap is copied first.

Examples:

```
> (define h (make-heap <=))

> (heap-add-all! h '(50 40 10 20 30))

> (for ([x (in-heap h)])
    (displayln x))
10
20
30
40
50

> (heap-count h)
5
```

(heap-sort! *v* <=?) → void?
  v : (and/c vector? (not/c immutable?))
  <=? : (-> any/c any/c any/c)

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

Examples:

```
> (define terms (vector "batch" "deal" "flock" "good
deal" "hatful" "lot"))

> (heap-sort! terms string<=?)

> terms
'#("batch" "deal" "flock" "good deal" "hatful" "lot")
```

8 Integer Sets

```
(require data/integer-set)
```

This library provides functions for working with finite sets of integers. This module is designed for sets that are compactly represented as groups of intervals, even when their cardinality is large. For example, the set of integers from -1000000 to 1000000 except for 0, can be represented as {[-1000000, -1], [1, 1000000]}. This data structure would not be a good choice for the set of all odd integers between 0 and 1000000, which would be {[1, 1], [3, 3], ... [999999, 999999]}.

In addition to the *integer set* abstract type, a *well-formed set* is a list of pairs of exact integers, where each pair represents a closed range of integers, and the entire set is the union of the ranges. The ranges must be disjoint and increasing. Further, adjacent ranges must have at least one integer between them. For example: `'((-1 . 2) (4 . 10))` is a well-formed-set as is `'((1 . 1) (3 . 3))`, but `'((1 . 5) (6 . 7))`, `'((1 . 5) (-3 . -1))`, `'((5 . 1))`, and `'((1 . 5) (3 . 6))` are not.

An integer set implements the stream and sequence generic interfaces.

```
(make-integer-set wfs) → integer-set?
  wfs : well-formed-set?
```

Creates an integer set from a well-formed set.

```
(integer-set-contents s) → well-formed-set?
  s : integer-set?
```

Produces a well-formed set from an integer set.

```
(set-integer-set-contents! s wfs) → void?
  s : integer-set?
  wfs : well-formed-set?
```

Mutates an integer set.

```
(integer-set? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is an integer set, `#f` otherwise.

```
(well-formed-set? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a well-formed set, `#f` otherwise.

```
(make-range) → integer-set?
(make-range elem) → integer-set?
  elem : exact-integer?
(make-range start end) → integer-set?
  start : exact-integer?
  end : exact-integer?
```

Produces, respectively, an empty integer set, an integer set containing only *elem*, or an integer set containing the integers from *start* to *end* inclusive, where (<= *start end*).

```
(intersect x y) → integer-set?
  x : integer-set?
  y : integer-set?
```

Returns the intersection of the given sets.

```
(subtract x y) → integer-set?
  x : integer-set?
  y : integer-set?
```

Returns the difference of the given sets (i.e., elements in *x* that are not in *y*).

```
(union x y) → integer-set?
  x : integer-set?
  y : integer-set?
```

Returns the union of the given sets.

```
(split x y) → integer-set? integer-set? integer-set?
  x : integer-set?
  y : integer-set?
```

Produces three values: the first is the intersection of *x* and *y*, the second is the difference *x* remove *y*, and the third is the difference *y* remove *x*.

```
(complement s start end) → integer-set?
  s : integer-set?
  start : exact-integer?
  end : exact-integer?
```

Returns a set containing the elements between *start* to *end* inclusive that are not in *s*, where (<= start-k end-k).

```
(symmetric-difference x y) → integer-set?
  x : integer-set?
  y : integer-set?
```

Returns an integer set containing every member of `x` and `y` that is not in both sets.

```
(member? k s) → boolean?
  k : exact-integer?
  s : integer-set?
```

Returns `#t` if `k` is in `s`, `#f` otherwise.

```
(get-integer set) → (or/c exact-integer? #f)
  set : integer-set?
```

Returns a member of `set`, or `#f` if `set` is empty.

```
(foldr proc base-v s) → any/c
  proc : (exact-integer? any/c . -> . any/c)
  base-v : any/c
  s : integer-set?
```

Applies `proc` to each member of `s` in ascending order, where the first argument to `proc` is the set member, and the second argument is the fold result starting with `base-v`. For example, `(foldr cons null s)` returns a list of all the integers in `s`, sorted in increasing order.

```
(partition s) → (listof integer-set?)
  s : (listof integer-set?)
```

Returns the coarsest refinement of the sets in `s` such that the sets in the result list are pairwise disjoint. For example, partitioning the sets that represent `'((1 . 2) (5 . 10))` and `'((2 . 2) (6 . 6) (12 . 12))` produces the a list containing the sets for `'((1 . 1) (5 . 5) (7 . 10))` `'((2 . 2) (6 . 6))`, and `'((12 . 12))`.

```
(count s) → exact-nonnegative-integer?
  s : integer-set?
```

Returns the number of integers in the given integer set.

```
(subset? x y) → boolean?
  x : integer-set?
  y : integer-set?
```

Returns true if every integer in `x` is also in `y`, otherwise `#f`.

9 Bit Vectors

```
(require data/bit-vector)
```

A *bit vector* is a mutable sequence whose elements are booleans. A bit vector also acts as a dictionary (`dict?` from `racket/dict`), where the keys are zero-based indexes and the values are the elements of the bit-vector. A bit-vector has a fixed size.

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

```
(make-bit-vector size [fill]) → bit-vector?
  size : exact-integer?
  fill : boolean? = #f
```

Creates a new bit-vector of size *size*. All elements are initialized to *fill*.

Examples:

```
> (bit-vector-ref (make-bit-vector 3) 2)
#f
> (bit-vector-ref (make-bit-vector 3 #t) 2)
#t
```

```
(bit-vector elem ...) → bit-vector?
  elem : boolean?
```

Creates a new bit-vector containing each *elem* in order.

Example:

```
> (bit-vector-ref (bit-vector #f #t #f) 1)
#t
```

```
(bit-vector? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a bit-vector, `#f` otherwise.

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

Returns the element at index *index*, if *index* is less than `(bit-vector-length bv)`. Otherwise, *default* is invoked if it is a procedure, returned otherwise.

Examples:

```
> (bit-vector-ref (bit-vector #f #t) 1)
#t
> (bit-vector-ref (bit-vector #f #t) 5 'not-there)
'not-there
```

```
(bit-vector-set! bv index value) → void?
  bv : bit-vector?
  index : (and/c exact-nonnegative-integer?
                 (</c (+ 1 (bit-vector-length vv))))
  value : boolean?
```

Sets the value at index *index* to be *value*.

Examples:

```
> (define bv (bit-vector #f #t))

> (bit-vector-ref bv 0)
#f
> (bit-vector-set! bv 0 #t)

> (bit-vector-ref bv 0)
#t
```

```
(bit-vector-length bv) → exact-nonnegative-integer?
  bv : bit-vector?
```

Returns the number of items in the bit-vector *bv*.

```
(bit-vector-popcount bv) → exact-nonnegative-integer?
  bv : bit-vector?
```

Returns the number of set bits in the bit-vector *bv*.

Example:

```
> (bit-vector-popcount (bit-vector #f #t #t))
2
```

```
(bit-vector-copy bv [start end]) → bit-vector?
  bv : bit-vector?
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (vector-length v)
```

Creates a fresh bit-vector with the same elements as *bv* from *start* (inclusive) to *end* (exclusive).

```
(in-bit-vector bv) → sequence?
  bv : bit-vector?
```

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

Examples:

```
> (define bv (bit-vector #f #t #f))

> (for/list ([x (in-bit-vector bv)]) x)
'(#f #t #f)
```

```
(for/bit-vector maybe-length (for-clause ...)
  body-or-break ... body)

maybe-length =
             | #:length length-expr
             | #:length length-expr #:fill fill-expr

  length-expr : exact-nonnegative-integer?
```

Iterates like `for/vector`, but results are accumulated into a bit-vector instead of a vector.

If the optional `#:length` clause is specified, the result of *length-expr* determines the length of the result bit-vector. In that case, the iteration can be performed more efficiently, and it terminates when the bit-vector is full or the requested number of iterations have been performed, whichever comes first. If *length-expr* specifies a length longer than the number of iterations, then the remaining slots of the vector are initialized to the value of *fill-expr*, which defaults to `#f` (i.e., the default argument of `make-bit-vector`).

Examples:

```
> (bit-vector->list
   (for/bit-vector ([i '(1 2 3)]) (odd? i)))
'(#t #f #t)
> (bit-vector->list
   (for/bit-vector #:length 2 ([i '(1 2 3)]) (odd? i)))
'(#t #f)
> (bit-vector->list
   (for/bit-vector #:length 4 ([i '(1 2 3)]) (odd? i)))
'(#t #f #t #f)
> (bit-vector->list
   (for/bit-vector #:length 4 #:fill #t ([i '(1 2 3)]) (odd? i)))
```

```
'(#t #f #t #t)
```

The `for/bit-vector` form may allocate a bit-vector and mutate it after each iteration of *body*, which means that capturing a continuation during *body* and applying it multiple times may mutate a shared bit-vector.

```
(for*/bit-vector maybe-length (for-clause ...)
  body-or-break ... body)
```

Like `for/bit-vector` but with the implicit nesting of `for*`.

```
(bit-vector->list bv) → (listof boolean?)
  bv : bit-vector?
(list->bit-vector bits) → bit-vector?
  bits : (listof boolean?)
(bit-vector->string bv) → (and/c string? #rx"^[01]*$")
  bv : bit-vector?
(string->bit-vector s) → bit-vector?
  s : (and/c string? #rx"^[01]*$")
```

Converts between bit-vectors and their representations as lists and strings.

Examples:

```
> (bit-vector->list (string->bit-vector "100111"))
'(#t #f #f #t #t #t)
> (bit-vector->string (list->bit-vector '(#t #f #t #t)))
"1011"
```

10 Union-Find: Sets with only Canonical Elements

```
(require data/union-find)
```

The union-find algorithm and data structure provides an API for representing sets that contain a single, canonical element. The sets support an (imperative) union operation (the library picks one of the canonical elements of the original sets to be the canonical element of the union), as well as getting and setting the canonical element.

These operations are not thread-safe.

```
(uf-new c) → uf-set?
  c : any/c
```

Makes a new set with the canonical element `c`.

This is a constant time operation.

Examples:

```
> (uf-new 'whale)
#<uf-set: 'whale>
> (uf-new 'dwarf-lantern)
#<uf-set: 'dwarf-lantern>
```

```
(uf-set? x) → boolean?
  x : any/c
```

Returns `#t` if `x` was created with `uf-new`, and `#f` otherwise.

This is a constant time operation.

Examples:

```
> (uf-set? (uf-new 'spiny-dogfish))
#t
> (uf-set? "I am not a uf-set")
#f
```

```
(uf-find a) → any/c
  a : uf-set?
```

Returns the canonical element of `a`.

This is an amortized (essentially) constant time operation.

Example:

```
> (uf-find (uf-new 'tasselled-wobbegong))
'tasselled-wobbegong
```

```
(uf-union! a b) → void?
  a : uf-set?
  b : uf-set?
```

Imperatively unifies `a` and `b`, making them both have the same canonical element. Either of `a` or `b`'s canonical elements may become the canonical element for the union.

This is an amortized (essentially) constant time operation.

Examples:

```
> (define a (uf-new 'sand-devil))

> (define b (uf-new 'pigeye))

> (uf-union! a b)

> (uf-find a)
'sand-devil
> (uf-find b)
'sand-devil
```

```
(uf-same-set? a b) → boolean?
  a : uf-set?
  b : uf-set?
```

Returns `#t` if the sets `a` and `b` have been unioned.

This is an amortized (essentially) constant time operation.

Examples:

```
> (define a (uf-new 'finetooth))

> (define b (uf-new 'speartooth))

> (uf-same-set? a b)
#f
> (uf-union! a b)

> (uf-same-set? a b)
#t
```

```
(uf-set-canonical! a c) → void?
  a : uf-set?
  c : any/c
```

Changes `a` to have a new canonical element.

This is an amortized (essentially) constant time operation.

Examples:

```
> (define a (uf-new 'sand-devil))

> (uf-set-canonical! a 'lemon)

> (uf-find a)
'lemon
> (define b (uf-new 'pigeye))

> (uf-union! a b)

> (uf-set-canonical! b 'sicklefin-lemon)

> (uf-find a)
'sicklefin-lemon
```