

Contents

1	Overview of hoard	4
2	Protocols	5
2.1	collection	5
2.1.1	Overview	5
2.1.2	API Reference	5
2.1.3	Examples	6
2.2	indexable	7
2.2.1	Overview	7
2.2.2	API Reference	7
2.2.3	Examples	8
2.3	extendable	9
2.3.1	Overview	9
2.3.2	API Reference	9
2.3.3	Examples	9
2.4	mutable	10
2.4.1	Overview	10
2.4.2	API Reference	10
2.4.3	Examples	10
2.5	enumerator	10
2.5.1	Overview	10
2.5.2	API Reference	10
2.5.3	Examples	11
2.6	enumerable	11
2.6.1	Overview	11
2.6.2	API Reference	12
2.6.3	Examples	15
2.7	collector	16
2.7.1	Overview	16
2.7.2	API Reference	16
2.7.3	Examples	18
2.8	comparator	18
2.8.1	Overview	18
2.8.2	API Reference	18
2.8.3	Examples	23
2.9	bag	23
2.9.1	Overview	23
2.9.2	API Reference	23

2.9.3	Examples	25
2.10	set	26
2.10.1	Overview	26
2.10.2	API Reference	26
2.10.3	Examples	29
2.11	queue	30
2.11.1	Overview	30
2.11.2	API Reference	30
2.11.3	Examples	32
2.12	deque	32
2.12.1	Overview	32
2.12.2	API Reference	32
2.12.3	Examples	35
2.13	stack	36
2.13.1	Overview	36
2.13.2	API Reference	36
2.13.3	Examples	37
2.14	priority-queue	38
2.14.1	Overview	38
2.14.2	API Reference	38
2.14.3	Examples	40
2.15	dictionary	41
2.15.1	Overview	41
2.15.2	API Reference	41
2.15.3	Examples	43
2.16	dictionary-enumerator	44
2.16.1	Overview	44
2.16.2	API Reference	45
2.16.3	Examples	46
2.17	dictionary-enumerable	46
2.17.1	Overview	46
2.17.2	API Reference	46
2.17.3	Examples	49
3	Implementations and Supporting Data Types	50
3.1	association	50
3.1.1	Overview	50
3.1.2	API Reference	50
3.1.3	Examples	51
3.2	range	51

3.2.1	Overview	51
3.2.2	API Reference	52
3.2.3	Examples	53
3.3	sorted-bag	53
3.3.1	Overview	53
3.3.2	API Reference	53
3.3.3	Examples	55
3.4	hash-bag	56
3.4.1	Overview	56
3.4.2	API Reference	56
3.4.3	Examples	59
3.5	sorted-set	60
3.5.1	Overview	60
3.5.2	API Reference	60
3.5.3	Examples	62
3.6	hash-set	63
3.6.1	Overview	63
3.6.2	API Reference	63
3.6.3	Examples	65
3.7	stretchy-vector	65
3.7.1	Overview	65
3.7.2	API Reference	66
3.7.3	Examples	70
3.8	contiguous-stack	70
3.8.1	Overview	70
3.8.2	API Reference	71
3.8.3	Examples	73
3.9	linked-stack	74
3.9.1	Overview	74
3.9.2	API Reference	74
3.9.3	Examples	75
3.10	contiguous-queue	76
3.10.1	Overview	76
3.10.2	API Reference	76
3.10.3	Examples	78
3.11	linked-queue	79
3.11.1	Overview	79
3.11.2	API Reference	79
3.11.3	Examples	81
3.12	ring-buffer	82

3.12.1	Overview	82
3.12.2	API Reference	82
3.12.3	Examples	85
3.13	linked-deque	86
3.13.1	Overview	86
3.13.2	API Reference	86
3.13.3	Examples	88
3.14	binary-heap	89
3.14.1	Overview	89
3.14.2	API Reference	89
3.14.3	Examples	92
3.15	pairing-heap	93
3.15.1	Overview	93
3.15.2	API Reference	93
3.15.3	Examples	95
3.16	sorted-dictionary	96
3.16.1	Overview	96
3.16.2	API Reference	96
3.16.3	Examples	99
4	Global Index	100
0.5		

1 Overview of hoard

hoard is a collections library for Bigloo. It consists of a number of generic protocols and corresponding concrete implementations. Currently, it supports generic protocols for abstract collections, dictionaries, sets, queues, stacks, and priority queues as well as generic protocols for enumeration (i.e., iteration) and comparison. Multiple concrete implementations of each protocol are provided. For example, there are both sorted/tree and hash based versions of sets and dictionaries.

Note: hoard heavily leverages bigloo-specific functionality, such as the object system, modules, and keyword arguments. Porting to other scheme systems would require significant effort.

2 Protocols

This chapter documents the protocols provided by hoard. Each protocol is described, the api documented, and examples given.

2.1 collection

2.1.1 Overview

The collection protocol provides the operations expected of an entity that represents an aggregate or container of other entities.

2.1.2 API Reference

plus 1 fill =collection? =/=object=/ [generic]

returns a boolean indicating whether or not `object` supports the collection protocol.

plus 1 fill =collection-length =/=object=/ [generic]

requires `object` support the `collection` protocol

returns the length of the `collection object`

plus 1 fill =collection-empty? =/=object=/ [generic]

requires `object` support the `collection` protocol

returns a boolean indicating whether the `collection object` is empty or not

plus 1 fill =collection-contains? =/=object itm=/ [generic]

requires `object` support the `collection` protocol and `itm` be an arbitrary value

returns a boolean indicating whether or not `itm` is in `object`

Note `collection-contains?` uses `equal?` for comparison, and for dictionary-like entities such as a hashtable or sorted-dictionary, it determines whether `itm` is a value in the collection, not whether it is a key in the collection. If you want to check to see if a key exists in such a collection, use `dictionary-contains?`

plus 1 fill `=collection-enumerator =/=object=/` [generic]

requires object support the collection protocol

returns an enumerator

note an `enumerator` provides the ability to enumerate or iterate through the items in a collection.

plus 1 fill `=collection-copy =/=object=/` [generic]

requires object support the collection protocol

returns a shallow copy of object

2.1.3 Examples

The standard Bigloo lists, vectors, strings, and hashtables are collections.

```
== (collection? (list 1 2 3)) => #t
(collection? (vector 1 2 3)) => #t
(collection? (create-hashtable)) => #t
(collection? "example") => #t
```

We can check to see if a collection is empty, obtain its length, and check to see if an item is contained within it.

```
== (collection-length (list 1 2 3)) => 3
(collection-empty? '()) => #t
(collection-empty? (list 1 2 3)) => #f
(collection-contains? (list 1 2 3) 3) => #t
```

When needed, we can also obtain a shallow copy of a collection.

```
== (let* ((vec1 (vector 1 2 3)) (vec2 (collection-copy vec1))) (and
(equal? vec1 vec2) (not (eq? vec1 vec2)))) => #t
```

The `collection-enumerator` function is used to implement the `enumerable` functionality. See Section [\[\[#anchor:enumerable\]\[2.6 \[enumerable\], page \]\]](#)

2.2 indexable

2.2.1 Overview

The indexable protocol specifies the operations expected of an indexable collection. An indexable collection is a collection which individual items can be referenced and potentially set via a key of some sort. This key can be an integer as is the case of lists, vectors, and strings, or arbitrary objects in the case of dictionary types.

2.2.2 API Reference

plus 1 fill =collection-indexable? =/=object=/ [generic]

returns a boolean indicating whether or not `object` supports the `indexable` protocol

plus 1 fill =collection-ref =/=object key [default-if-not-found]=/ [generic]

requires `object` support the `indexable` protocol, `key` be an arbitrary value, and optionally, `default-if-not-found` be an arbitrary value

returns the value at the given `key` or throws `&invalid-index-exception` unless `default-if-not-found` is provided in which case `default-if-not-found` is returned

note For vectors, lists, and their ilk, `key` is an integer value representing the position in the collection. For dictionaries, `key` is the traditional key associated with such types.

plus 1 fill =collection-set! =/=object key value=/ [generic]

requires `object` support the `indexable` protocol and both `key` and `value` be arbitrary values **returns** `#unspecified` or throws `&unsupported-operation-exception` if `collection-set!` is not supported.

modifies `object` so that value at `key` is now `value` A generic function that sets the value found at `key` to the given value. If the

key is not valid for the collection, `&invalid-index-exception` is thrown.

plus 1 fill =collection-slice =/=object keys=/ [generic]

requires object support the `indexable` protocol and `keys` be an object implementing the `enumerable` protocol

returns an `enumerator` providing access to the elements in the collection `object` represented by the provided enumerable of `keys`.

2.2.3 Examples

The standard Bigloo lists, vectors, strings, and hashtables are indexable.

```
== (collection-indexable? (list 1 2 3)) => #t
(collection-indexable? (vector 1 2 3)) => #t
(collection-indexable? "test string") => #t
(collection-indexable? (create-hashtable)) => #t
```

For collections that are position addressable such as lists, vectors, and strings as well as other, you can reference each individual element with `collection-ref` given its position as a key.

```
== (collection-ref (list 1 2 3) 1) => 2
(collection-ref (vector 1 2 3) 2) => 3
(collection-ref "test string" 0) => t
(collection-ref (vector 1 2 3) 3 -1) => -1
(collection-ref (vector 1 2 3) 3) => &invalid-index-exception
```

For dictionary-like collections, such as a hashtables, `collection-ref` obtains the value in the collection associated with the provided `key`.

```
== (let ((table (create-hashtable))) (hashtable-put! 'a 1) (hashtable-put! 'b 2) (hashtable-put! 'c 3) (collection-ref table 'b)) => 2
(let ((table (create-hashtable))) (hashtable-put! 'a 1) (hashtable-put! 'b 2) (hashtable-put! 'c 3) (collection-ref table 'd -1)) => -1
(let ((table (create-hashtable))) (hashtable-put! 'a 1) (hashtable-put! 'b 2) (hashtable-put! 'c 3) (collection-ref table 'd)) => &invalid-index-exception
```

It is sometimes useful to obtain a subset of elements provided by an indexable collection. `collection-slice` provides this functionality.

```
== (let ((slice-enumer (collection-slice (iota 50) (range :start 15 :end 30)))) (enumerable-collect slice-enumer list-collector)) => (15 16 17 18 19 20 21 22 23 24 25 26 27 28 29)
```



```
(enumerable-collect (collection-slice (vector 3 4 5 6) '(2 3)) list-collector)
⇒ (5 6)
```

2.3 extendable

2.3.1 Overview

The extendable protocol specifies the operations expected of a collection that can be dynamically extended beyond its initially allocated size.

2.3.2 API Reference

plus 1 fill =collection-extendable? =/=object=/ [generic]

returns a boolean indicating whether or not **object** is extendable.

plus 1 fill =collection-extend! =/=object value=/ [generic]

requires **object** support the **extendable** protocol and **value** be an arbitrary value. **returns** **#unspecified** or throws **&unsupported-operation-exception** if **collection-extend!** is not supported.

modifies **object** by adding **value**.

note for dictionary-like collections **value** should be an **association** representing the key and value.

2.3.3 Examples

Bigloo lists and hashtables (as well as a number of other collection types) implement the **extendable** protocol:

```
== (collection-extendable? (list 1 2 3)) ⇒ #t
(collection-extendable? (create-hashtable)) ⇒ #t
Both can be extended:
== (collection-extend! (list 1 2 3 4) 5) ⇒ (5 1 2 3 4)
(let ((table (hashtable :comparator string-comparator (= > "a" 1) (= >
"b" 2)))) (collection-extend! table (= > "c" 3)) (enumerable-collect table
list-collector)) ⇒ (1 2 3)
```

2.4 mutable

2.4.1 Overview

The `mutable` protocol specifies the operations expected of an mutable collection. A mutable collection is one in which the collection can be modified in place.

2.4.2 API Reference

plus 1 fill `=collection-mutable? =/=object=/` [generic]

returns a boolean indicating whether or not `object` is mutable

2.4.3 Examples

Not surprisingly, the standard Bigloo lists, vectors, and hashtables are mutable.

```
== (collection-mutable? (list)) => #t  
(collection-mutable? (vector 1 2 3)) => #t  
(collection-mutable? "test string") => #t
```

2.5 enumerator

2.5.1 Overview

The `enumerator` protocol is an abstraction for enumerating or iterating through the items contained by an object. These items may be pre-existing or generated on demand. Collections are the most common object to provide `enumerators` but other objects, such as interval ranges, can provide enumerators as well.

2.5.2 API Reference

plus 1 fill `=enumerator? =/=object=/` [generic]

returns a boolean indicating whether or not `object` supports the `enumerator` protocol

plus 1 fill `=enumerator-move-next! =/=object=/` [generic]

requires `object` support the `enumerator` protocol

returns a boolean indicating whether or not `object` additional items available

modifies `object` such that upon returning `#t` the next item under enumeration is current. Otherwise, `object` remains unmodified.

note `enumerator-move-next!` must be called before `enumerator-current`. If it is not, `&invalid-state-exception` is thrown.

plus 1 fill `=enumerator-current` `=/=object=/` [generic]

requires `object` support the `enumerator` protocol

returns the item found at the `enumerators` current position or if `enumerator-move-next!` has not been called before, `&invalid-state-exception` is thrown.

plus 1 fill `=enumerator-copy` `=/=object=/` [generic]

requires `object` support the `enumerator` protocol

returns a copy of the `enumerator`.

2.5.3 Examples

You seldom use `enumerator` directly but, instead, use the procedures and macros provided by `enumerable`. However, if needed you can use the `enumerator` protocol directly.

```
== (let ((enumer (enumerable-enumerator (list 1 2 3 4 5)))) (let loop
((cont (enumerator-move-next! enumer))) (when cont (print (enumerator-
current enumer)) (loop (enumerator-move-next! enumer))))) + 1 + 2
+ 3 + 4 + 5 => #unspecified
```

2.6 enumerable

2.6.1 Overview

`enumerable` is a protocol implemented by those objects that support a notion of enumeration or iteration. By providing an `enumerator`, they gain support for mapping, folding, filtering, and more.

2.6.2 API Reference

plus 1 fill =enumerable? =/=object=/ [generic]

returns a boolean indicating whether or not `object` supports the `enumerable` protocol

plus 1 fill =enumerable-enumerator =/=obj=/ [generic]

requires `object` support the `enumerable` protocol

returns an `enumerator`

plus 1 fill =enumerable-for-each =/=proc enum1 enum2 ...=/ [syntax]

requires `enum=s` to either support the `=enumerable` or `enumerator` protocol and `proc` be a procedure with an arity compatible with applying it element-wise to the elements provided by the `enum=s`.
`enum=s`.\\ \\ ***returns*** `=#unspecified`

note `enumerable-for-each` is applied for the side-effects resulting from applying `proc`.

plus 1 fill =enumerable-map =/=proc enum1 enum2 ...=/ [syntax]

requires `enum=s` to either support the `=enumerable` or `enumerator` protocol and `proc` be a procedure with an arity compatible with applying it element-wise to the elements provided by the `enum=s`.
`enum=s`.\\ \\ ***returns*** an `=enumerator` providing the results of applying `proc`

plus 1 fill =enumerable-filter =/=predicate enum=/ [syntax]

requires `enum` to either support the `enumerable` or `enumerator` protocol and `predicate` be a procedure returning a boolean indicating whether or not an element should be retained.

returns an `enumerator` providing the retained elements

plus 1 fill =enumerable-take-while =/=predicate enum=/ [syntax]

requires `enum` to either support the `enumerable` or `enumerator` protocol and `predicate` be a procedure returning a boolean indicating whether or not an elements should continue to be consumed.

returns an `enumerator` providing the consumed elements

plus 1 fill `=enumerable-take =/n enum=/` [syntax]

requires `enum` to either support the `enumerable` or `enumerator` protocol and `n` be an integer representing the max number of items to take

returns an `enumerator` providing upto `n` elements

plus 1 fill `=enumerable-fold =/proc seed enum1 enum2 ...=/` [syntax]

requires `enum=s` to either support the `=enumerable` or `enumerator` protocol, `proc` to be a procedure compatible with applying it to a seed and the element-wise elements provided by the `enum=s` (it should produce a new seed value), and `=seed` should be an appropriate initial seed value for `proc`.

returns the result of folding `proc` over the provided `enum=s`\\
\\ **note** `=proc` is applied left to right with the seed resulting from the previous application being used in the next. When no more elements are available, the final seed is returned.

plus 1 fill `=enumerable-any? =/predicate enum1 enum2 ...=/` [syntax]

requires `enum=s` to either support the `=enumerable` or `enumerator` protocol and `predicate` to be a procedure compatible with applying it element-wise to the elements provided by the `enum=s`, returning a boolean.\\
\\ **returns** a boolean indicating whether or not any of the elements are `=#t` for `predicate`.

plus 1 fill `=enumerable-every? =/predicate enum1 enum2 ...=/` [syntax]

requires `enum=s` to either support the `=enumerable` or `enumerator` protocol and `predicate` to be a procedure compatible with applying it element-wise to the elements provided by the `enum=s`, returning a boolean.\\ \\ **returns** a boolean indicating whether or not all of the elements are `=#t` for `predicate`.

plus 1 fill `=enumerable-skip-while` `=/predicate` `enum1` `enum2` ...`=/` [syntax]

requires `enum=s` to either support the `=enumerable` or `enumerator` protocol and `predicate` to be a procedure compatible with applying it element-wise to the elements provided by the `enum=s`, returning a boolean indicating whether those elements should be skipped.\\ \\ **returns** an `=enumerator` providing access to the elements in the original `enum=s` after those that were skipped according to `=predicate`.

plus 1 fill `=enumerable-skip` `=/n` `enum1` `enum2` ...`=/` [syntax]

requires `enum=s` to either support the `=enumerable` or `enumerator` protocol and `n` to be an integer representing the number of elements to skip from `enum=s`.\\ \\ **returns** an `=enumerator` providing access to the elements in the original `=enum=s` after those that were skipped.

plus 1 fill `=enumerable-append` `=/enum1` `enum2` ...`=/` [syntax]

requires `enum=s` to either support the `=enumerable` or `enumerator` protocol.

returns an `enumerator` providing access to the elements in the original `=enum=s` appended left to right.

plus 1 fill `=enumerable-collect` `=/enum` `collector``=/` [syntax]

requires `enum` to either support the `enumerable` or `enumerator` protocol and `collector` to support the `collector` protocol.

returns a value collected from the given `enum` according to provided `collector`.

note `enumerable-collect` is commonly used to transform the values provided by `enumerators` into concrete collections, although their uses are more flexible than that. For example, to collect all of the values from an `enumerator` into a list you could do the following:

```
== (enumerable-collect enumer list-collector) See Section [[#anchor:collector][2.7 [collector], page ]] for more information.
```

2.6.3 Examples

All of the built in Bigloo collection types are `=enumerable=s`.

```
== (enumerable? (list 1 2 3)) => #t
(enumerable? (vector 1 2 3)) => #t
(enumerable? "test string") => #t
(enumerable? (create-hashtable)) => #t
```

With `numerable-for-each`, it is possible to iterate over any enumerable.

A few examples follow:

```
== (let ((count 0)) (enumerable-for-each (lambda (x) (set! count (+ x count))) (vector 1 2 3)) count)
=> 6
```

```
(enumerable-for-each (lambda (x) (print x)) (range :start 1 :end 5))
+ 1 + 2 + 3 + 4 => #unspecified
```

It is also possible to map a function over `=enumerable=s`.

```
== (enumerable-collect (enumerable-map (lambda (x) (+ x 1)) (list 1 2 3 4)) list-collector)
=> (2 3 4 5)
```

For dictionary type collections, the values are mapped over.

```
== (let ((table (hashtable (=> 'a 1) (=> 'b 2) (=> 'c 3)))) (print (enumerable-collect (enumerable-map (lambda (x) (+ x 1)) table) vector-collector)))
```

Given an appropriate seed and procedure, `enumerable-fold` can be used to reduce an enumerable to a single value.

```
== (enumerable-fold (lambda (s v) (+ s v)) 0 (range :start 1 :end 6))
=> 15
```

Filtering of values is also supported.

```
== (enumerable-collect (enumerable-filter odd? (range :start 1 :end 10)) list-collector)
=> (1 3 5 7 9)
```

It is also possible to consume values while a predicate remains true.

```
== (enumerable-collect (enumerable-take-while (lambda (x) (< x 5)) (range :start 0 :end 10)) list-collector)
```

```
⇒ (0 1 2 3 4)
```

Or to test whether any or all values match a give predicate.

```
== (enumerable-any? odd? (list 4 6 8 7))
```

```
⇒ #t
```

```
(enumerable-any? odd? (list 4 6 8 10))
```

```
⇒ #f
```

```
(enumerable-every? even? (list 2 4 6 8))
```

```
⇒ #t
```

```
(enumerable-every? even? (list 2 4 6 7))
```

```
⇒ #f
```

As shown in the above examples, `enumerable-collect` can be used to gather the values of an enumerable into a new collection, but it is more general than that. In fact, it is a general reduction facility. For example, the sum of an `enumerable` can be obtained as follows:

```
== (enumerable-collect (range :start 1 :end 6) sum-collector)
```

```
⇒ 15
```

For full details, See Section [\[\[#anchor:collector\]\[2.7 \[collector\], page \]\]](#).

2.7 collector

2.7.1 Overview

The `collector` protocol defines the methods required for a general reduction facility for `enumerable=s`. Examples include `=collection` conversion and various accumulations. It is modeled after the `Collector` interface of Java.

2.7.2 API Reference

plus 1 fill `=collector? =/=object=/` [generic]

returns a boolean indicating whether or not `object` supports the `collector` protocol

plus 1 fill `=collector-supplier =/=object=/` [generic]

requires `object` support the `collector` protocol

returns a thunk that when called returns an object for collecting results

plus 1 fill =collector-accumulate =/=coll supp val=/ [generic]

requires coll support the **collector** protocol, **supp** be an object originally created via the thunk returned by **collector-supplier**, and **val** be an arbitrary item to be accumulated.

returns supp updated with val

plus 1 fill =collector-combine =/=coll suppa suppb=/ [generic]

requires coll support the **collector** protocol and **suppa** and **suppb** be objects originally created via the thunk returned by **collector-supplier**,

returns a single entity combining both **suppa** and **suppb**

plus 1 fill =collector-finish =/=coll supp=/ [generic]

requires coll support the **collector** protocol and **supp** be an object originally created via the thunk returned by **collector-supplier**,

returns a single entity from the accumulated **supp** possibly with a final transformation.

plus 1 fill **make-collector** =/:supplier :accumulate :combine :finish=/ [procedure]

requires **:supplier** be a thunk returning an object used for accumulation by the collector, **:accumulate** be a procedure taking the current accumulation object and a val and returning an updated accumulation object, **:combine** be a procedure taking 2 accumulation objects and returning an accumulation object combining the 2, and **:finish** be a procedure taking the final accumulation object and returning an possibly transformed result.

returns a final accumulation object.

plus 1 fill =variable =/=+list-collector+=/ [Variable]

+list-collector+ is a collector that accumulates the items provided by an **enumerator** into a list.

plus 1 fill =variable =/+stretchy-vector-collector+=/ [Variable]

+stretchy-vector-collector+ is a collector that accumulates the items provided by an **enumerator** into a stretchy-vector.

plus 1 fill =variable =/+vector-collector+=/ [Variable]

+vector-collector+ is a collector that accumulates the items provided by an **enumerator** into a vector.

plus 1 fill =variable =/+sum-collector+=/ [Variable]

+sum-collector+ is a collector that sums the items provided by an **enumerator**.

2.7.3 Examples

To create your own version of a list **collector** is as simple as the following:

```
== (define my-list-collector (make-collector :supplier (lambda () '())
:accumulate (lambda (supp val) (cons val supp)) :combine (lambda (a b)
(append a b)) :finish (lambda (x) (reverse x))))
```

To obtain the product of all numbers in an **enumerable**

```
== (define product-collector (make-collector :supplier (lambda () 1)
:accumulate (lambda (supp val) (* supp val)) :combine (lambda (a b) (*
a b)) :finish (lambda (x) x)))
```

2.8 comparator

2.8.1 Overview

The **comparator** protocol defines those methods required to support comparison and, optionally, total ordering of a specific type. Although not identical, it is very similar to the functionality found in SRFI-128.

2.8.2 API Reference

plus 1 fill =comparator? =/=object=/ [generic]

returns a boolean indicating whether or not **object** supports the **comparator** protocol

plus 1 fill =comparator-ordered? =/=comp=/ [generic]

requires `comp` implement the `comparator` protocol.

returns a boolean indicating whether or not `comp` supports ordering.

plus 1 fill =comparator-hashable? =/=comp=/ [generic]

requires `comp` implement the `comparator` protocol.

returns a boolean indicating whether or not `comp` supports hashing.

plus 1 fill =comparator-type? =/=comp val=/ [generic]

requires `comp` implement the `comparator` protocol and `val` be an arbitrary object.

returns a boolean indicating whether or not `val` is of an appropriate type for `comp`

plus 1 fill `comparator?` =/=comp a b=/ [generic]

requires `comp` implement the `comparator` protocol and `a` and `b` be of the type supported by `comp`.

returns a boolean indicating whether or not `a` is equal to `b`

note `comparator?` is supported by all `comparator` instances.

plus 1 fill =comparator<? =/=comp a b=/ [generic]

requires `comp` implement the `comparator` protocol and `a` and `b` be of the type supported by `comp`.

returns a boolean indicating whether or not `a` is less than `b`

note `comparator<?` is only supported by `comparator` instances that are ordered.

plus 1 fill `=comparator>? =/=comp a b=/` [generic]

requires `comp` implement the `comparator` protocol and `a` and `b` be of the type supported by `comp`.

returns a boolean indicating whether or not `a` is greater than `b`

note `comparator>?` is only supported by `comparator` instances that are ordered.

plus 1 fill `comparator<? =/=comp a b=/` [generic]

requires `comp` implement the `comparator` protocol and `a` and `b` be of the type supported by `comp`.

returns a boolean indicating whether or not `a` is less than or equal to `b`

note `comparator<?=` is only supported by `comparator` instances that are ordered.

plus 1 fill `comparator>?= =/=comp a b=/` [generic]

requires `comp` implement the `comparator` protocol and `a` and `b` be of the type supported by `comp`.

returns a boolean indicating whether or not `a` is greater than or equal to `b`

note `comparator>?=` is only supported by `comparator` instances that are ordered.

plus 1 fill `=comparator-hash =/=comp val=/` [generic]

requires `comp` implement the `comparator` protocol and `val` of the type supported by `comp`.

returns an integer hash value

note `comparator-hash` is only supported by `comparator` instances that are hashable.

plus 1 fill **make-comparator** =/:type? :equal? [:less-than?] [:hash]=/ [procedure]

requires **:type?** be a single argument procedure returning a boolean indicating whether or not the argument is of the type supported by the **comparator**, **:equal?** be a 2 argument procedure returning whether or not the arguments are equal, **:less-than?** be a 2 argument procedure returning whether or not the first argument is less than or equal to the second, and **:hash** be a single argument procedure returning an appropriate integer value.

returns a **comparator**

note **:less-than?** and **:hash** are optional but at least one needs to be provided. Obviously, both can also be provided. If **:less-than?** is not provided, the comparator will not be ordered. Similarly, if the **:hash** is not provided, the comparator will not be hashable.

plus 1 fill =variable =/+number-comparator+=/ [Variable]

+number-comparator+ is a comparator instance for numbers. It is ordered and hashable.

plus 1 fill =variable =/+char-comparator+=/ [Variable]

+char-comparator+ is a comparator instance for characters. It is ordered and hashable. The ordering is case-sensitive.

plus 1 fill =variable =/+char-ci-comparator+=/ [Variable]

+char-ci-comparator+ is a comparator instance for characters. It is ordered and hashable. The ordering is case-insensitive.

plus 1 fill =variable =/+ucs2-comparator+=/ [Variable]

+ucs2-comparator+ is a comparator instance for unicode characters. It is ordered and hashable. The ordering is case-sensitive.

plus 1 fill =variable =/+ucs2-ci-comparator+=/ [Variable]

+ucs2-ci-comparator+ is a comparator instance for unicode characters. It is ordered and hashable. The ordering is case-insensitive.

plus 1 fill =variable =/+string-comparator+=/ [Variable]

+string-comparator+ is a comparator instance for strings. It is ordered and hashable. The ordering is case-sensitive.

plus 1 fill =variable =/+string-ci-comparator+=/ [Variable]

+string-ci-comparator+ is a comparator instance for strings. It is ordered and hashable. The ordering is case-insensitive.

plus 1 fill =variable =/+symbol-comparator+=/ [Variable]

+symbol-comparator+ is a comparator instance for symbols. It is ordered and hashable. The ordering is case-sensitive.

plus 1 fill =variable =/+symbol-ci-comparator+=/ [Variable]

+symbol-ci-comparator+ is a comparator instance for symbols. It is ordered and hashable. The ordering is case-insensitive.

plus 1 fill =variable =/+ucs2-string-comparator+=/ [Variable]

+ucs2-string-comparator+ is a comparator instance for unicode strings. It is ordered and hashable. The ordering is case-sensitive.

plus 1 fill =variable =/+ucs2-string-ci-comparator+=/ [Variable]

+ucs2-string-ci-comparator+ is a comparator instance for unicode strings. It is ordered and hashable. The ordering is case-insensitive.

plus 1 fill =variable =/+keyword-comparator+=/ [Variable]

+keyword-comparator+ is a comparator instance for keywords. It is ordered and hashable. The ordering is case-sensitive.

plus 1 fill =variable =/+keyword-ci-comparator+=/ [Variable]

+keyword-ci-comparator+ is a comparator instance for keywords. It is ordered and hashable. The ordering is case-insensitive.

2.8.3 Examples

To demonstrate the use of the comparator protocol, we will use the `+number-comparator+` instance.

`+number-comparator+` is both ordered and hashable:

```
== (comparator-ordered? number-comparator) => #t
```

```
(comparator-hashable? number-comparator) => #t
```

All comparators must support type checking and equality:

```
== (comparator-type? number-comparator 4) => #t
```

```
(comparator-type? number-comparator "dog") => #f
```

```
(comparator=? number-comparator 4 4) => #t
```

```
(comparator=? number-comparator 3 4) => #f
```

Being ordered, we can use the ordered comparison methods with `number-comparator`:

```
== (comparator<? number-comparator 4 5) => #t
```

```
(comparator>? number-comparator 4.0 5) => #f
```

```
(comparator<=? number-comparator 5 5) => #t
```

```
(comparator>=? number-comparator 5 6.0) => #f
```

And being hashable, we can use the hash method:

```
== (comparator-hash comp 4) => 4
```

```
(comparator-hash comp 5.0) => 5
```

2.9 bag

2.9.1 Overview

The `bag` protocol defines those methods required by a bag implementation. A bag is set-like data structure that can contain multiple copies of an item.

2.9.2 API Reference

plus 1 fill `=bag? =/=object=/` [generic]

returns a boolean indicating whether or not `object` supports the `bag` protocol

plus 1 fill `=bag-copy =/=bag=/` [generic]

requires `bag` implement the bag protocol.

returns a shallow copy of `bag`.

plus 1 fill `=bag-empty? =/=bag=/` [generic]

requires `bag` implement the bag protocol.

returns a boolean indicating whether or not `bag` contains any items.

plus 1 fill `=bag-insert! =/=bag item=/` [generic]

requires `bag` implement the bag protocol, and `item` be an arbitrary object.

modifies `bag` so that it contains a copy or an additional copy, if one already exists, of `item`.

returns unspecified

plus 1 fill `=bag-delete! =/=bag item=/` [generic]

requires `bag` implement the bag protocol, and `item` be an arbitrary object.

modifies `bag` so that it contains one less copy of `item`. If 0 copies of `item` exist it is removed from `bag`.

returns unspecified

plus 1 fill `=bag-contains? =/=bag item=/` [generic]

requires `bag` implement the bag protocol, and `item` be an arbitrary object.

returns a boolean indicating whether or not the `bag` contains `item`.

plus 1 fill `=bag-count =/=bag item=/` [generic]

requires `bag` implement the bag protocol, and `item` be an arbitrary object.

returns the number of `item`s found in `=bag`.

plus 1 fill =bag-count-set! =/=bag item count=/ [generic]

requires **bag** implement the bag protocol, **item** be an arbitrary object, and **count** be an integer representing the number of **item**=s to include in =bag

modifies **bag** to contain **count** number of **items** unless **count** is less than or equal to 0 which results in the all **item**=s **begin** removed from =bag.

returns #unspecified

plus 1 fill =bag-length =/=bag=/ [generic]

requires **bag** implement the bag protocol

returns the number of items contained in **bag**

2.9.3 Examples

2 different implementations of the bag protocol are currently provided by hoard. One is tree-based, and the other is hash-based. To create a tree-based bag use:

```
== (let ((bag (make-sorted-bag :comparator number-comparator))) (bag? bag))  
⇒ #t
```

The **:comparator** argument must be an object implementing the comparator protocol for the type of item to be stored in the bag. The comparator must be ordered.

To create a hash-based bag use:

```
== (let ((bag (make-hash-bag :comparator number-comparator))) (bag? bag))  
⇒ #t
```

The **:comparator** argument, like in the tree-based example, must be an object implementing the comparator protocol for the type of item to be stored in the bag. However, the comparator must be hashable not ordered.

Assuming a bag has been created as above, to insert an item into a bag use:

```
== (bag-insert! bag 1) ⇒ #unspecified
```

The count of an item can be obtained with:

```
== (bag-count bag 1) ⇒ 1  
(bag-count bag 2) ⇒ 0
```

And the count can be explicitly set using:

```
== (bag-count-set! bag 1 4) => #unspecified
```

```
(bag-count bag 1) => 4
```

Or a single copy deleted with:

```
== (bag-delete! bag 1) => #unspecified
```

```
(bag-count bag 1) => 3
```

To check to see if a bag is empty:

```
== (bag-empty? bag) => #f
```

To check whether an item is a member of a bag:

```
== (bag-contains? bag 1) => #t
```

```
(bag-contains? bag 3) => #f
```

And finally, to make a copy:

```
== (let ((bag2 (bag-copy? bag))) (eq? bag bag2)) => #f
```

In addition, **sorted-bag** implements the **enumerable** protocol. See Section [\[\[#anchor:enumerable-Examples\]\[2.6.3 \[enumerable Examples\], page \]\]](#).

2.10 set

2.10.1 Overview

The **set** protocol defines those methods required by a set implementation. A set is a collection of objects where each object is unique.

2.10.2 API Reference

```
plus 1 fill =set? =/=object=/ [generic]
```

returns a boolean indicating whether or not **object** supports the **set** protocol

```
plus 1 fill =set-copy =/=set=/ [generic]
```

requires **set** implement the set protocol.

returns a shallow copy of **set**.

```
plus 1 fill =set-empty? =/=set=/ [generic]
```

requires **set** implement the set protocol.

returns a boolean indicating whether or not **set** contains any items.

plus 1 fill =set-insert! =/=set item=/ [generic]

requires **set** implement the set protocol, and **item** be an arbitrary object.

modifies **set** so that it contains a copy or an additional copy, if one already exists, of **item**.

returns unspecified

plus 1 fill =set-delete! =/=set item=/ [generic]

requires **set** implement the set protocol, and **item** be an arbitrary object.

modifies **set** so that it contains one less copy of **item**. If 0 copies of **item** exist it is removed from **set**.

returns unspecified

plus 1 fill =set-contains? =/=set item=/ [generic]

requires **set** implement the set protocol, and **item** be an arbitrary object.

returns a boolean indicating whether or not the **set** contains **item**.

plus 1 fill =set-length =/=set=/ [generic]

requires **set** implement the set protocol

returns the number of items contained in **set**

plus 1 fill =set-union! =/=set . sets=/ [generic]

requires **set** implement the set protocol and **sets** be an arbitrary number of additional objects implementing the set protocol.

modifies **set** to be the union of all sets provided. **returns** #unspecified

plus 1 fill =set-union /=set . sets= / [generic]

requires **set** implement the set protocol and **sets** be an arbitrary number of additional objects implementing the set protocol.

returns a new set of the same type as **set** containing the union of all sets provided.

plus 1 fill =set-intersect! /=set . sets= / [generic]

requires **set** implement the set protocol and **sets** be an arbitrary number of additional objects implementing the set protocol.

modifies **set** to be the intersection of all sets provided. **returns** #unspecified

plus 1 fill =set-intersect /=set . sets= / [generic]

requires **set** implement the set protocol and **sets** be an arbitrary number of additional objects implementing the set protocol.

returns a new set of the same type as **set** containing the intersection of all sets provided.

plus 1 fill =set-difference! /=set . sets= / [generic]

requires **set** implement the set protocol and **sets** be an arbitrary number of additional objects implementing the set protocol.

modifies **set** to be the difference of all sets provided. **returns** #unspecified

plus 1 fill =set-difference /=set . sets= / [generic]

requires `set` implement the set protocol and `sets` be an arbitrary number of additional objects implementing the set protocol.

returns a new set of the same type as `set` containing the difference of all sets provided.

2.10.3 Examples

2 different implementations of the set protocol are currently provided by `hoard`. One is tree-based, and the other is hash-based. To create a tree-based set use:

```
== (let ((set (make-sorted-set :comparator number-comparator))) (set? set))  
⇒ #t
```

The `:comparator` argument must be an object implementing the comparator protocol for the type of item to be stored in the set. The comparator must be ordered.

To create a hash-based set use:

```
== (let ((set (make-hash-set :comparator number-comparator))) (set? set))  
⇒ #t
```

The `:comparator` argument, like in the tree-based example, must be an object implementing the comparator protocol for the type of item to be stored in the set. However, the comparator must be hashable not ordered.

Assuming a set has been created, to insert an item into a set use:

```
== (set-insert! set 1) ⇒ #unspecified
```

Or deleted with:

```
== (set-delete! set 1) ⇒ #unspecified
```

```
(set-contains? set 1) ⇒ #f
```

To check to see if a set is empty:

```
== (set-empty? set) ⇒ #f
```

To check whether an item is a member of a set:

```
== (set-contains? set 1) ⇒ #t
```

```
(set-contains? set 3) ⇒ #f
```

To make a copy:

```
== (let ((set2 (set-copy? set))) (eq? set set2)) ⇒ #f
```

To non-destructively perform the union, intersection, and difference:

```
== (let* ((set1 (sorted-set :comparator number-comparator 1 2)) (set2  
(hash-set :comparator number-comparator 2 3)) (uset (set-union set1 set2))  
(iset (set-intersect set1 set2)) (dset (set-difference set1 set2)))
```

(list (enumerable-collect uset ~~list-collector~~)) (enumerable-collect iset
~~list-collector~~) (enumerable-collect dset ~~list-collector~~))

2.11 queue

2.11.1 Overview

The **queue** protocol defines the methods required by a queue implementation. A queue is a first-in-first-out sequential data structure.

2.11.2 API Reference

plus 1 fill =queue? =/=object=/ [generic]

returns a boolean indicating whether or not **object** supports the **queue** protocol

plus 1 fill =queue-copy =/=queue=/ [generic]

requires **queue** implement the queue protocol.

returns a shallow copy of **queue**.

plus 1 fill =queue-empty? =/=queue=/ [generic]

requires **queue** implement the queue protocol.

returns a boolean indicating whether or not **queue** contains any items.

plus 1 fill =queue-first =/=queue=/ [generic]

requires **queue** implement the queue protocol.

returns returns the first item in **queue**. If **queue** is empty, a **&invalid-state-exception** is thrown.

plus 1 fill =queue-enqueue! =/=queue item=/ [generic]

requires `queue` implement the queue protocol and `item` be an arbitrary item

returns returns `unspecified`. If `queue` has a fixed capacity and is full, a `&invalid-state-exception` is thrown.

modifies `queue` by placing `item` on the end.

plus 1 fill `=queue-dequeue! =/=queue=/` [generic]

requires `queue` implement the queue protocol.

returns returns the first item in `queue`. If `queue` is empty, a `&invalid-state-exception` is thrown.

modifies `queue` by removing the first item.

plus 1 fill `=queue-length =/=queue=/` [generic]

requires `queue` implement the queue protocol.

returns returns the the number of items in `queue`

plus 1 fill `=queue-fixed-capacity? =/=queue=/` [generic]

requires `queue` implement the queue protocol.

returns returns a boolean indicating whether or not `queue` has a fixed capacity.

plus 1 fill `=queue-capacity =/=queue=/` [generic]

requires `queue` implement the queue protocol.

returns returns the capacity (i.e., the number of items that it can hold) of `queue` or `unspecified` if it has no fixed capacity.

2.11.3 Examples

hoard provides linked and contiguous implementations of the `queue` protocol.

```
== (queue? (contiguous-queue :capacity 5)) => #t (queue? (linked-queue)) => #t
```

You can add items to a queue:

```
== (let ((q (linked-queue))) (queue-enqueue! q 1) (queue-enqueue! q 2) (enumerable-collect q list-collector)) => (1 2)
```

look at the first item:

```
== (let ((q (linked-queue))) (queue-enqueue! q 1) (queue-enqueue! q 2) (queue-first q)) => 1
```

or remove the first item:

```
== (let ((q (linked-queue))) (queue-enqueue! q 1) (queue-enqueue! q 2) (queue-dequeue! q) (queue-front q)) => 2
```

It is also possible to test to see if a queue is empty:

```
== (queue-empty? (linked-queue)) => #t (queue-empty? (linked-queue 1 2 3)) => #f
```

A copy of a queue can be useful:

```
== (let* ((q1 (linked-queue 1 2 3)) (q2 (queue-copy q1))) (eq? q1 q2)) => #f
```

Some queue implementations have a fixed-capacity:

```
== (queue-fixed-capacity? (contiguous-queue :capacity 5)) => #t (queue-fixed-capacity? (linked-queue)) => #f (queue-capacity (contiguous-queue :capacity 5)) => 5 (queue-capacity (linked-queue)) => #unspecified
```

2.12 deque

2.12.1 Overview

The `deque` protocol defines the methods required by a deque implementation. A deque is a sequential data structure similar to a queue but allowing insertion and removal of items from both the front and back.

2.12.2 API Reference

plus 1 fill =deque? =/=object=/ [generic]

returns a boolean indicating whether or not `object` supports the `deque` protocol

plus 1 fill =deque-copy =/=deque=/ [generic]

requires `deque` implement the deque protocol.

returns a shallow copy of `deque`.

plus 1 fill `=deque-empty? =/=deque=/` [generic]

requires `deque` implement the deque protocol.

returns a boolean indicating whether or not `deque` contains any items.

plus 1 fill `=deque-first =/=deque=/` [generic]

requires `deque` implement the deque protocol.

returns returns the first item in `deque`. If `deque` is empty, a `&invalid-state-exception` is thrown.

plus 1 fill `=deque-last =/=deque=/` [generic]

requires `deque` implement the deque protocol.

returns returns the last item in `deque`. If `deque` is empty, a `&invalid-state-exception` is thrown.

plus 1 fill `=deque-enqueue! =/=deque item=/` [generic]

requires `deque` implement the deque protocol and `item` be an arbitrary item

returns returns `unspecified`. If `deque` has a fixed capacity and is full, a `&invalid-state-exception` is thrown.

modifies `deque` by placing `item` on the end.

plus 1 fill `=deque-enqueue-front! =/=deque item=/` [generic]

requires `deque` implement the deque protocol and `item` be an arbitrary item

returns returns `unspecified`. If `deque` has a fixed capacity and is full, a `&invalid-state-exception` is thrown.

modifies `deque` by placing `item` on the front.

plus 1 fill `=deque-dequeue! /=deque=/` [generic]

requires `deque` implement the deque protocol.

returns returns the first item in `deque`. If `deque` is empty, a `&invalid-state-exception` is thrown.

modifies `deque` by removing the first item.

plus 1 fill `=deque-dequeue-back! /=deque=/` [generic]

requires `deque` implement the deque protocol.

returns returns the last item in `deque`. If `deque` is empty, a `&invalid-state-exception` is thrown.

modifies `deque` by removing the last item.

plus 1 fill `=deque-length /=deque=/` [generic]

requires `deque` implement the deque protocol.

returns returns the the number of items in `deque`

plus 1 fill `=deque-fixed-capacity? /=deque=/` [generic]

requires `deque` implement the deque protocol.

returns returns a boolean indicating whether or not `deque` has a fixed capacity.

plus 1 fill =deque-capacity /=deque= [generic]

requires deque implement the deque protocol.

returns returns the capacity (i.e., the number of items that it can hold) of **deque** or **unspecified** if it has no fixed capacity.

2.12.3 Examples

hoard provides double-linked and contiguous implementations of the **deque** protocol.

```
== (deque? (ring-buffer :capacity 5)) => #t (deque? (linked-deque))  
=> #t
```

You can add items to both the front and back of a deque:

```
== (let ((q (linked-deque))) (deque-enqueue! q 2) (deque-enqueue! q  
3) (deque-enqueue-front! q 1) (enumerable-collect q list-collector)) =>  
(1 2 3)
```

look at the first or last item:

```
== (let ((q (linked-deque))) (deque-enqueue! q 1) (deque-enqueue! q  
2) (deque-enqueue! q 3) (cons (deque-first q) (deque-last q)) => (1 . 3)
```

or remove the first and last items:

```
== (let ((q (linked-deque))) (deque-enqueue! q 1) (deque-enqueue! q  
2) (deque-enqueue! q 3) (list (deque-dequeue! q) (deque-dequeue-back! q)  
(deque-front q)) => (1 3 2)
```

It is also possible to test to see if a deque is empty:

```
== (deque-empty? (linked-deque)) => #t  
(deque-empty? (linked-deque 1 2 3)) => #f
```

A copy of a deque can be useful:

```
== (let* ((q1 (linked-deque 1 2 3)) (q2 (deque-copy q1))) (eq? q1 q2))  
=> #f
```

Some deque implementations have a fixed-capacity:

```
== (deque-fixed-capacity? (ring-buffer :capacity 5)) => #t  
(deque-fixed-capacity? (linked-deque)) => #f  
(deque-capacity (ring-buffer :capacity 5)) => 5  
(deque-capacity (linked-deque)) => #unspecified
```

2.13 stack

2.13.1 Overview

The **stack** protocol defines the methods required by a stack implementation. A stack is a last-in-first-out sequential data structure.

2.13.2 API Reference

plus 1 fill `=stack? =/=object=` [generic]

returns a boolean indicating whether or not **object** supports the **stack** protocol

plus 1 fill `=stack-copy =/=stack=` [generic]

requires **stack** implement the stack protocol.

returns a shallow copy of **stack**.

plus 1 fill `=stack-empty? =/=stack=` [generic]

requires **stack** implement the stack protocol.

returns a boolean indicating whether or not **stack** contains any items.

plus 1 fill `=stack-top =/=stack=` [generic]

requires **stack** implement the stack protocol.

returns returns the top item on **stack**. If **stack** is empty, a `&invalid-state-exception` is thrown.

plus 1 fill `=stack-push! =/=stack item=` [generic]

requires **stack** implement the stack protocol and **item** be an arbitrary item

returns returns **unspecified**. If **stack** has a fixed capacity and is full, a `&invalid-state-exception` is thrown.

modifies **stack** by placing **item** on the top.

plus 1 fill =stack-pop! /=stack=/ [generic]

requires **stack** implement the stack protocol.

returns returns the top item on **stack**. If **stack** is empty, a **&invalid-state-exception** is thrown.

modifies **stack** by removing the top item.

plus 1 fill =stack-length /=stack=/ [generic]

requires **stack** implement the stack protocol.

returns returns the the number of items on **stack**

plus 1 fill =stack-fixed-capacity? /=stack=/ [generic]

requires **stack** implement the stack protocol.

returns returns a boolean indicating whether or not **stack** has a fixed capacity.

plus 1 fill =stack-capacity /=stack=/ [generic]

requires **stack** implement the stack protocol.

returns returns the capacity (i.e., the number of items that it can hold) of **stack** or **unspecified** if it has no fixed capacity.

2.13.3 Examples

hoard provides linked and contiguous implementations of the **stack** protocol.

```
== (stack? (contiguous-stack :capacity 5)) => #t (stack? (linked-stack)) => #t
```

You can push items on a stack:

```
== (let ((s (linked-stack))) (stack-push! s 1) (stack-push! s 2) (enumerable-collect s list-collector)) => (2 1)
```

look at the top item:

```
== (let ((s (linked-stack))) (stack-push! s 1) (stack-push! s 2) (stack-top s)) => 2
```

or remove the top item:

```
== (let ((s (linked-stack))) (stack-push! q 1) (stack-push! q 2) (stack-
pop! q) (stack-top q)) => 1
```

It is also possible to test to see if a stack is empty:

```
== (stack-empty? (linked-stack)) => #t
(stack-empty? (linked-stack 1 2 3)) => #f
```

A copy of a stack can be useful:

```
== (let* ((q1 (linked-stack 1 2 3)) (q2 (stack-copy q1))) (eq? q1 q2))
=> #f
```

Some stack implementations have a fixed-capacity:

```
== (stack-fixed-capacity? (contiguous-stack :capacity 5)) => #t
(stack-fixed-capacity? (linked-stack)) => #f
(stack-capacity (contiguous-stack :capacity 5)) => 5
(stack-capacity (linked-stack)) => #unspecified
```

2.14 priority-queue

2.14.1 Overview

The `priority-queue` protocol defines the methods required by a priority-queue implementation. A priority-queue is a sequential data structure where items are ordered by priority.

2.14.2 API Reference

plus 1 fill `=priority-queue? =/=object=/` [generic]

returns a boolean indicating whether or not `object` supports the `priority-queue` protocol

plus 1 fill `=priority-queue-copy =/=priority-queue=/` [generic]

requires `priority-queue` implement the priority-queue protocol.

returns a shallow copy of `priority-queue`.

plus 1 fill `=priority-queue-empty? =/=priority-queue=/` [generic]

requires `priority-queue` implement the priority-queue protocol.

returns a boolean indicating whether or not **priority-queue** contains any items.

plus 1 fill **=priority-queue-first** **!=priority-queue** **!=** [generic]

requires **priority-queue** implement the priority-queue protocol.

returns returns the first item in **priority-queue**. If **priority-queue** is empty, a **&invalid-state-exception** is thrown.

plus 1 fill **=priority-queue-enqueue!** **!=priority-queue item** **!=** [generic]

requires **priority-queue** implement the priority-queue protocol and **item** be an arbitrary item

returns returns **unspecified**. If **priority-queue** has a fixed capacity and is full, a **&invalid-state-exception** is thrown.

modifies **priority-queue** by placing **item** in the queue according to its priority.

plus 1 fill **=priority-queue-dequeue!** **!=priority-queue** **!=** [generic]

requires **priority-queue** implement the priority-queue protocol.

returns returns the first item in **priority-queue**. If **priority-queue** is empty, a **&invalid-state-exception** is thrown.

modifies **priority-queue** by removing the first item.

plus 1 fill **=priority-queue-length** **!=priority-queue** **!=** [generic]

requires **priority-queue** implement the priority-queue protocol.

returns returns the the number of items in **priority-queue**

plus 1 fill =priority-queue-fixed-capacity? =/priority-queue=/ [generic]

requires `priority-queue` implement the priority-queue protocol.

returns returns a boolean indicating whether or not `priority-queue` has a fixed capacity.

plus 1 fill =priority-queue-capacity =/priority-queue=/ [generic]

requires `priority-queue` implement the priority-queue protocol.

returns returns the capacity (i.e., the number of items that it can hold) of `priority-queue` or `unspecified` if it has no fixed capacity.

2.14.3 Examples

hoard provides linked and contiguous implementations of the `priority-queue` protocol.

```
== (priority-queue? (binary-heap :capacity 5 :comparator number-comparator))  
⇒ #t (priority-queue? (pairing-heap :comparator number-comparator))  
⇒ #t
```

You can add items to a priority-queue:

```
== (let ((q (pairing-heap :comparator number-comparator))) (priority-queue-enqueue! q 1) (priority-queue-enqueue! q 2) (enumerable-collect q list-collector)) ⇒ (1 2)
```

look at the first item:

```
== (let ((q (pairing-heap :comparator number-comparator))) (priority-queue-enqueue! q 2) (priority-queue-enqueue! q 1) (priority-queue-first q)) ⇒ 1
```

or remove the first item:

```
== (let ((q (pairing-heap :comparator number-comparator))) (priority-queue-enqueue! q 1) (priority-queue-enqueue! q 2) (priority-queue-dequeue! q) (priority-queue-front q)) ⇒ 2
```

It is also possible to test to see if a priority-queue is empty:

```
== (priority-queue-empty? (pairing-heap :comparator number-comparator))  
⇒ #t
```



```

(priority-queue-empty? (pairing-heap :comparator number-comparator))
⇒ #f
A copy of a priority-queue can be useful:
== (let* ((q1 (pairing-heap :comparator number-comparator 1 2 3))
(q2 (priority-queue-copy q1))) (eq? q1 q2)) ⇒ #f
Some priority-queue implementations have a fixed-capacity:
== (priority-queue-fixed-capacity? (binary-heap :capacity 5 :compara-
tor number-comparator)) ⇒ #t
(priority-queue-fixed-capacity? (pairing-heap :comparator number-comparator))
⇒ #f
(priority-queue-capacity (binary-heap :capacity 5 :comparator number-comparator))
⇒ 5
(priority-queue-capacity (pairing-heap :comparator number-comparator))
⇒ #unspecified

```

2.15 dictionary

2.15.1 Overview

The `dictionary` protocol defines the methods required by a dictionary implementation. A dictionary is a data structure maintaining associations between keys and values.

2.15.2 API Reference

plus 1 fill `=dictionary? =/=object=/` [generic]

returns a boolean indicating whether or not `object` supports the `dictionary` protocol

plus 1 fill `=dictionary-copy =/=dict=/` [generic]

requires `dict` implement the dictionary protocol.

returns a shallow copy of `dict`

plus 1 fill `=dictionary-empty? =/=dict=/` [generic]

requires `dict` implement the dictionary protocol.

returns a boolean indicating whether or not the dictionary contains any associations.

plus 1 fill =dictionary-get /=dict key=/ [generic]

requires dict implement the dictionary protocol and **key** be an arbitrary object

returns the value associated with **key** or **#f** if no such association exists.

plus 1 fill =dictionary-put! /=dict key value=/ [generic]

requires dict implement the dictionary protocol and **key** and **value** be arbitrary objects

modifies dict so that it contains the association of **key** to **value**. If an association with **key** already exists it is replaced, and if not, a new association is created.

returns #unspecified

plus 1 fill =dictionary-length /=dict=/ [generic]

requires dict implement the dictionary protocol.

returns the number of items in **dict**.

plus 1 fill =dictionary-remove! /=dict key=/ [generic]

requires dict implement the dictionary protocol and **key** be an arbitrary object.

modifies dict by removing the association with the key **key**.

returns #unspecified

plus 1 fill =dictionary-contains? /=dict key=/ [generic]

requires dict implement the dictionary protocol and **key** be an arbitrary object.

returns a boolean indicating whether or not **dict** contains an association with the key **key**.

plus 1 fill =dictionary-update! =/=dict key value exist-fun=/ [generic]

requires `dict` implement the dictionary protocol, `key` and `value` be arbitrary objects, and `exist-fun` which is a procedure expecting a single value and returning an updated value when an association with a key `key` already exists.

modifies `dict` such that if the dictionary doesn't currently contain an association with a key `key`, it contains an association with key `key` and value `value`, or if an existing association exists updates it so that has the value obtained by applying `exist-fun` to its value.

returns unspecified

plus 1 fill =dictionary-enumerator =/=dict=/ [generic]

requires `dict` implement the dictionary protocol.

returns an object implementing the `dictionary-enumerator` protocol allowing the enumeration of the associations contained in `dict`.

2.15.3 Examples

Bigloo's native hashtable and the hoard provided sorted-dictionary implement the `dictionary` protocol.

```
== (dictionary? (create-hashtable)) => #t
```

```
(dictionary? (sorted-dictionary :comparator number-comparator))
```

Checking whether a dictionary is empty:

```
== (dictionary-empty? (create-hashtable)) => #t
```

```
(dictionary-empty? (hashtable (=> "A" 1))) => #f
```

Associations can be added to a dictionary:

```
== (let ((dict (create-hashtable))) (dictionary-put! dict "a" 1) (dictionary-put! dict "b" 2) (map (lambda (kv) (cons (=>key kv) (=>value kv))) (dictionary-enumerable-collect dict list-collector))) => (("a" . 1) ("b" . 2))
```

And removed (assuming the insertions above):

```
== (dictionary-remove! dict "a") (map (lambda (kv) (cons (=>key kv)
(=>value kv))) (dictionary-enumerable-collect dict list-collector)) => (("b"
. 2))
```

The value associated with a given key is easily obtained:

```
== (let ((dict (hashtable (=> "a" 1) (=> "b" 2) (=> "c" 3)))) (dictionary-
get dict "b") => 2
```

The **dictionary-update!** method can be used to update an existing value or insert a new value if an existing association does not exist:

```
== (let ((dict (hashtable (=> "a" 1) (=> "b" 2) (=> "c" 3)))) (dictionary-
update! dict "b" 0 (lambda (x) (+ x 1))) (dictionary-update! dict "d" 0
(lambda (x) (+ x 1))) (dictionary-get dict "b") => 3 (dictionary-get dict
"d") => 0
```

To obtain the number of associations in a dictionary, call **dictionary-length**:

```
== (let ((dict (hashtable (=> "a" 1) (=> "b" 2) (=> "c" 3)))) (dictionary-
length dict) => 3
```

Querying whether an association with a given key is accomplished with **dictionary-contains?**:

```
== (let ((dict (hashtable (=> "a" 1) (=> "b" 2) (=> "c" 3)))) (dictionary-
contains? dict "a") => #t
(dictionary-contains? dict "d") => #f
```

Copying a dictionary is accomplished with **dictionary-copy**:

```
== (let* ((dict1 (hashtable (=> "a" 1) (=> "b" 2) (=> "c" 3))) (dict2
(dictionary-copy dict1))) (eq? dict1 dict2) => #f (and (dictionary-contains? dict1
"a") (dictionary-contains? dict2 "a")) => #t
```

And last, but not least, you can obtain a **dictionary-enumerator** to enumerate the elements of the dictionary:

```
== (let ((dict (hashtable (=> "a" 1) (=> "b" 2) (=> "c" 3)))) (dictionary-
enumerator? (dictionary-enumerator dict)) => #t
```

2.16 dictionary-enumerator

2.16.1 Overview

The **dictionary-enumerator** protocol is an abstraction for enumerating or iterating through the items contained by an object supporting the **dictionary** protocol. These items may be pre-existing or generated on demand. It is very similar to the **enumerator** protocol but provides access to both the keys and values of the dictionary.

2.16.2 API Reference

plus 1 fill =dictionary-enumerator? /=object=/ [generic]

returns a boolean indicating whether or not `object` supports the `dictionary-enumerator` protocol

plus 1 fill =dictionary-enumerator-move-next! /=object=/ [generic]

requires `object` support the `dictionary-enumerator` protocol

returns a boolean indicating whether or not `object` has additional items available

modifies `object` such that upon returning `#t` the next item under enumeration is current. Otherwise, `object` remains unmodified.

note `dictionary-enumerator-move-next!` must be called before `dictionary-enumerator-current`. If it is not, `&invalid-state-exception` is thrown.

plus 1 fill =dictionary-enumerator-current /=object=/ [generic]

requires `object` support the `dictionary-enumerator` protocol

returns the association found at the `dictionary-enumerators` current position or if `dictionary-enumerator-move-next!` has not been called before, `&invalid-state-exception` is thrown.

plus 1 fill =dictionary-enumerator-key /=object=/ [generic]

requires `object` support the `dictionary-enumerator` protocol

returns the key found at the `dictionary-enumerators` current position or if `dictionary-enumerator-move-next!` has not been called before, `&invalid-state-exception` is thrown.

plus 1 fill =dictionary-enumerator-value /=object=/ [generic]

requires object support the `dictionary-enumerator` protocol

returns the value found at the `dictionary-enumerators` current position or if `dictionary-enumerator-move-next!` has not been called before, `&invalid-state-exception` is thrown.

plus 1 fill `=dictionary-enumerator-copy =/=object=/` [generic]

requires object support the `dictionary-enumerator` protocol

returns a copy of the `dictionary-enumerator`.

2.16.3 Examples

You seldom use `dictionary-enumerator` directly but, instead, use the procedures and macros provided by `dictionary-enumerable`. However, if needed you can use the `dictionary-enumerator` protocol directly.

```
== (let ((enumerator (enumerable-dictionary-enumerator (hashtable :com-
parator number-comparator (= > 'a 1) (= > 'b 2) (= > 'c 3)))) (let loop
((cont (dictionary-enumerator-move-next! enumerator))) (when cont (print
(dictionary-enumerator-value enumerator)) (loop (dictionary-enumerator-move-
next! enumerator))))) 1 2 3 => #unspecified
```

2.17 dictionary-enumerable

2.17.1 Overview

`dictionary-enumerable` is a protocol implemented by those dictionary objects that support a notion of enumeration or iteration. By providing an `enumerator`, they gain support for mapping, folding, filtering, and more.

2.17.2 API Reference

plus 1 fill `=dictionary-enumerable? =/=object=/` [generic]

returns a boolean indicating whether or not `object` supports the `dictionary-enumerable` protocol

plus 1 fill `=dictionary-enumerable-enumerator =/=obj=/` [generic]

requires object support the `dictionary-enumerable` protocol

returns an `dictionary-enumerator`

plus 1 fill `=dictionary-enumerable-for-each =/proc enum=/` [syntax]

requires `enum` to either support the `dictionary-enumerable` or `dictionary-enumerator` protocol and `proc` be a procedure that accepts two arguments, a key and value.

returns `#unspecified`

note `dictionary-enumerable-for-each` is applied for the side-effects resulting from applying `proc`.

plus 1 fill `=dictionary-enumerable-map =/proc enum=/` [syntax]

requires `enum` to either support the `dictionary-enumerable` or `dictionary-enumerator` protocol and `proc` be a procedure taking the key and value of an association and returning a new association.

returns an `enumerator` providing the results of applying `proc`

plus 1 fill `=dictionary-enumerable-filter =/predicate enum=/` [syntax]

requires `enum` to either support the `dictionary-enumerable` or `enumerator` protocol and `predicate` be a procedure accepting a key and a value and returning a boolean indicating whether or not the association should be retained.

returns an `dictionary-enumerator` providing the retained elements

plus 1 fill `=dictionary-enumerable-fold =/proc seed enum=/` [syntax]

requires `enum` to either support the `dictionary-enumerable` or `dictionary-enumerator` protocols, `proc` to be a procedure compatible with applying it to a seed and a key and value provided by `enum` (it should produce a new seed value), and `seed` should be an appropriate initial seed value for `proc`.

returns the result of folding `proc` over the provided `enum`

note `proc` is applied left to right with the seed resulting from the previous application being used in the next. When no more elements are available, the final seed is returned.

plus 1 fill `=dictionary-enumerable-any? /=predicate enum=/` [syntax]

requires `enum` to either support the `dictionary-enumerable` or `dictionary-enumerator` protocols and `predicate` to be a procedure accepting a key and a value and returning a boolean.

returns a boolean indicating whether or not any of the associations are `#t` for `predicate`.

plus 1 fill `=dictionary-enumerable-every? /=predicate enum=/` [syntax]

requires `enum` to either support the `dictionary-enumerable` or `dictionary-enumerator` protocols and `predicate` to be a procedure accepting a key and a value and returning a boolean.

returns a boolean indicating whether or not all of the associations are `#t` for `predicate`.

plus 1 fill `=dictionary-enumerable-append /=enum1 enum2 ...=/` [syntax]

requires `enum=s` to either support the `=dictionary-enumerable` or `dictionary-enumerator` protocols.

returns an `dictionary-enumerator` providing access to the elements in the original `=enum=s` appended left to right.

plus 1 fill =dictionary-enumerable-collect =/enum collector=/ [syntax]

requires `enum` to either support the `dictionary-enumerable` or `dictionary-enumerator` protocol and `collector` to support the `collector` protocol.

returns a value collected from the given `enum` according to the provided `collector`.

note `dictionary-enumerable-collect` is commonly used to transform the keys and values provided by `dictionary-enumerator=s` into concrete collections, although their uses are more flexible than that. For example, to collect all of the values from an `=dictionary-enumerator` into a list you could do the following:

`== (dictionary-enumerable-collect enumer list-collector)` See Section [\[\[#anchor:collector\]\[2.7 \[collector\], page \]\]](#) for more information.

2.17.3 Examples

Bigloo's hashtables and hoard's sorted-dictionary are `=dictionary-enumerable=s`.

`== (dictionary-enumerable? (create-hashtable)) => #t`
`(dictionary-enumerable? (sorted-dictionary :comparator number-comparator))`

With `numerable-for-each`, it is possible to iterate over any dictionary-enumerable. A few examples follow:

`== (let ((count 0)) (dictionary-enumerable-for-each (lambda (k v) (set! count (+ v count))) (hashtable (=> 'a 1) (=> 'b 2) (=> 'c 3))) count)`
`=> 6`

It is also possible to map a function over `=dictionary-enumerable=s`.

`== (map =>value (dictionary-enumerable-collect (dictionary-enumerable-map (lambda (k v) (=> k (+ v 1))) (hashtable (=> 'a 1) (=> 'b 2) (=> 'c 3))) list-collector)) => (2 3 4)`

Given an appropriate seed and procedure, `dictionary-enumerable-fold` can be used to reduce a dictionary-enumerable to a single value.

`== (dictionary-enumerable-fold (lambda (s k v) (+ s v)) 0 (hashtable (=> 'a 1) (=> 'b 2) (=> 'c 3) (=> 'd 4) (=> 'e 5)))`
`=> 15`

Filtering of values is also supported.

```
== (map =>value (dictionary-enumerable-collect (dictionary-enumerable-  
filter (lambda (k v) (odd? v)) (hashtable (=> 'a 1) (=> 'b 2) (=> 'c 3)  
(=> 'd 4) (=> 'e 5))) list-collector)) => (2 3 4 5)
```

Or to test whether any or all associations match a give predicate.

```
== (dictionary-enumerable-any? (lambda (k v) (odd? v)) (hashtable  
(=> 'a 1) (=> 'b 2) (=> 'c 3) (=> 'd 4) (=> 'e 5))) => #t  
(dictionary-enumerable-any? (lambda (k v) (odd? v)) (hashtable (=>  
'a 2) (=> 'b 4) (=> 'c 6) (=> 'd 8) (=> 'e 10))) => #f  
(dictionary-enumerable-every? (lambda (k v) (even? v)) (hashtable (=>  
'a 2) (=> 'b 4) (=> 'c 6) (=> 'd 8) (=> 'e 10))) => #t  
(dictionary-enumerable-every? (lambda (k v) (even? v)) (hashtable (=>  
'a 2) (=> 'b 3) (=> 'c 6) (=> 'd 8) (=> 'e 10))) => #f
```

As shown in the above examples, `dictionary-enumerable-collect` can be used to gather the values of an dictionary-enumerable into a new collection, but it is more general than that. In fact, it is a general reduction facility. For full details, See Section [\[\[#anchor:collector\]\[2.7 \[collector\], page \]\]](#).

3 Implementations and Supporting Data Types

Each of the collection implementations and supporting data types are documented here.

3.1 association

3.1.1 Overview

association is a simple data structure containing a key and a value. It is used by the **dictionary** protocol.

3.1.2 API Reference

plus 1 fill `=association? =/=object=/` [procedure]

returns a boolean indicating whether or not **object** is an association.

plus 1 fill `==> =/=key value=/` [procedure]

requires **key** and **value** be arbitrary objects.

returns a new association with key **key** and value **value**.

plus 1 fill ==>key /=assoc=/ [procedure]

requires `assoc` be an association

returns the key of the association.

plus 1 fill ==>value /=assoc=/ [procedure]

requires `assoc` be an association

returns the value of the association.

plus 1 fill =pair->association /=kv-pair=/ [procedure]

requires `kv-pair` be a pair with the car the key and the cdr the value

returns an association with key the car of `kv-pair` and the value the cdr of `kv-pair`.

3.1.3 Examples

`association?` tests whether an object is an **association** or not:

```
== (association? '(key . value)) => #f
```

```
(association? (=> 'key 'value)) => #t
```

A new association is constructed with `=>`:

```
== (let ((assoc (=> 'key 'value))) (=>key assoc)) => 'key
```

A pair consisting of a key and value can also be converted to an **association**:

```
== (let ((assoc (pair->association '(key . value)))) (=>key assoc)) =>
'key
```

The key of an association is obtained with `=>key`:

```
== (=>key (=> "key" 1)) => "key"
```

The value of an association is obtained with `=>value`:

```
== (=>value (=> "key" 1)) => 1
```

3.2 range

3.2.1 Overview

range is a data structure representing an integer interval having a start and an end (exclusive) possessing values between the two at a given step. It supports methods for iterating and mapping over the specified interval.

3.2.2 API Reference

`range` implements the `enumerable` protocol. See Section [\[\[#anchor:enumerable\]\[2.6 \[enumerable\], page \]\]](#).

plus 1 fill `=range? =/object=/` [procedure]

returns a boolean indicating whether or not `object` is a range.

plus 1 fill `range =/[:start 0] :end [:step 1]=/` [procedure]

requires `:start`, `:end`, and `:step` be integers with `:start <= :end` when `:step` is positive or `:start >= :end` when `:step` is negative. `:step` can not be 0. `:start` and `:step` are optional and default to 0 and 1, respectively.

returns a range with the specified `start`, `end`, and `step`.

plus 1 fill `=range-for-each =/proc range=/` [procedure]

requires `proc` be a procedure compatible with applying it element-wise to the values represented by `range` and `range` be a valid instance of the range data type.

returns `#unspecified`.

note `range-for-each` is applied for the side-effects resulting from applying `proc`.

plus 1 fill `=range-map =/proc range=/` [procedure]

requires `proc` be a procedure compatible with applying it element-wise to the values represented by `range` and `range` be a valid instance of the range data type.

returns a list providing the results of applying `proc` to each integer in `range`

3.2.3 Examples

Creating a **range** is straight forward:

```
== (range :end 5) ; [0...4] (range :start 2 :end 5) ; [2,3,4] (range :start
2 :end 10 :step 2) ; [2,4,6,8]
```

Once you have a **range**, you can iterate and map over it:

```
== (range-for-each print (range :end 5)) → 0 → 1 → 2 → 3 → 4
⇒ #unspecified
(range-map (lambda (x) (+ x 1)) (range :end 5)) ⇒ (1 2 3 4 5)
```

In addition, **range** implements the **enumerable** protocol. See Section [\[\[#anchor:enumerable-Examples\]\[2.6.3 \[enumerable Examples\], page \]\]](#).

3.3 sorted-bag

3.3.1 Overview

sorted-bag is an implementation of **bag**. It is based on a balanced-binary tree and keeps the elements of the bag sorted.

3.3.2 API Reference

sorted-bag implements the **bag**, **collection**, **extendable**, **mutable**, and **enumerable** protocols. See Section [\[\[#anchor:bag\]\[2.9 \[bag\], page \]\]](#). See Section [\[\[#anchor:collection\]\[2.1 \[collection\], page \]\]](#). See Section [\[\[#anchor:extendable\]\[2.3 \[extendable\], page \]\]](#). See Section [\[\[#anchor:mutable\]\[2.4 \[mutable\], page \]\]](#). See Section [\[\[#anchor:enumerable\]\[2.6 \[enumerable\], page \]\]](#).

plus 1 fill **=sorted-bag? =/=object=** [procedure]

returns a boolean indicating whether or not **object** is a sorted-bag.

plus 1 fill **make-sorted-bag =/:comparator=** [procedure]

requires **:comparator** be an object implementing the **comparator** protocol. **comparator** must be ordered.

returns an instance of **sorted-bag**

plus 1 fill **sorted-bag =/:comparator . items=** [procedure]

requires : `comparator` be an object implementing the `comparator` protocol and `items` be a list of objects for which `comparator` is applicable. `comparator` must be ordered.

returns an instance of `sorted-bag`

modifies the returned `sorted-bag` so that it contains `items`

plus 1 fill `=sorted-bag-copy /=bag=/` [procedure]

requires `bag` be a `sorted-bag`

returns a shallow copy of `bag`.

plus 1 fill `=sorted-bag-empty? /=bag=/` [procedure]

requires `bag` be a `sorted-bag`

returns a boolean indicating whether or not `bag` contains any items.

plus 1 fill `=sorted-bag-insert! /=bag item=/` [procedure]

requires `bag` be a `sorted-bag`, and `item` be an arbitrary object supported by the comparator used to create `bag`.

modifies `bag` so that it contains a copy or an additional copy, if one already exists, of `item`.

returns unspecified

plus 1 fill `=sorted-bag-delete! /=bag item=/` [procedure]

requires `bag` be a `sorted-bag`, and `item` be an arbitrary object supported by the comparator used to create `bag`.

modifies `bag` so that it contains one less copy of `item`. If 0 copies of `item` exist it is removed from `bag`.

returns unspecified

plus 1 fill =sorted-bag-contains? /=bag item=/ [procedure]

requires **bag** be a sorted-bag, and **item** be an arbitrary object supported by the comparator used to create **bag**

returns a boolean indicating whether or not the **bag** contains **item**.

plus 1 fill =sorted-bag-count /=bag item=/ [procedure]

requires **bag** be a sorted-bag, and **item** be an arbitrary object supported by the comparator used to create **bag**

returns the number of **item=s** found in **=bag**.

plus 1 fill =sorted-bag-count-set! /=bag item count=/ [procedure]

requires **bag** be a sorted-bag, **item** be **item** be an arbitrary object supported by the comparator used to create **bag**, and **count** be an integer representing the number of **item=s** to include in **=bag**

modifies **bag** to contain **count** number of **items** unless **count** is less than or equal to 0 which results in the all **item=s** begin removed from **=bag**.

returns #unspecified

plus 1 fill =bag-length /=bag=/ [procedure]

requires **bag** be a sorted-bag

returns the number of items contained in **bag**

3.3.3 Examples

2 procedures are provided for creating a **sorted-bag**. The first creates an empty bag and the other populates the bag with the items passed to it:

```
== (enumerable-collect (make-sorted-bag :comparator number-comparator)  
list-collector) => ()
```

```
(enumerable-collect (sorted-bag :comparator number-comparator 1 1 1
3) list-collector)
⇒ (1 1 1 3)
```

Assuming a sorted-bag has been created as above, to insert an item into a sorted-bag use:

```
== (sorted-bag-insert! bag 1) ⇒ #unspecified
```

The count of an item can be obtained with:

```
== (sorted-bag-count bag 1) ⇒ 1
```

```
(sorted-bag-count bag 2) ⇒ 0
```

And the count can be explicitly set using:

```
== (sorted-bag-count-set! bag 1 4) ⇒ #unspecified
```

```
(sorted-bag-count bag 1) ⇒ 4
```

Or a single copy deleted with:

```
== (sorted-bag-delete! bag 1) ⇒ #unspecified
```

```
(sorted-bag-count bag 1) ⇒ 3
```

To check to see if a sorted-bag is empty:

```
== (sorted-bag-empty? bag) ⇒ #f
```

To check whether an item is a member of a sorted-bag:

```
== (sorted-bag-contains? bag 1) ⇒ #t
```

```
(sorted-bag-contains? bag 3) ⇒ #f
```

And finally, to make a copy:

```
== (let ((bag2 (sorted-bag-copy? bag))) (eq? bag bag2)) ⇒ #f
```

sorted-bag also implements the **bag**, **collection**, **mutable**, and **enumerable** protocols. See Section [\[\[#anchor:bag-Examples\]\[2.9.3 \[bag Examples\], page \]\]](#). See Section [\[\[#anchor:collection-Examples\]\[2.1.3 \[collection Examples\], page \]\]](#). See Section [\[\[#anchor:extendable-Examples\]\[2.3.3 \[extendable Examples\], page \]\]](#) See Section [\[\[#anchor:mutable-Examples\]\[2.4.3 \[mutable Examples\], page \]\]](#). See Section [\[\[#anchor:enumerable-Examples\]\[2.6.3 \[enumerable Examples\], page \]\]](#).

3.4 hash-bag

3.4.1 Overview

hash-bag is an implementation of bag. As its name would imply, it is a hashtable-based implementation.

3.4.2 API Reference

hash-bag implements the **bag**, **collection**, **extendable**, **mutable**, and **enumerable** protocols. See Section [\[\[#anchor:bag\]\[2.9 \[bag\], page \]\]](#). See

Section [\[\[#anchor:collection\]\[2.1 \[collection\], page \]\]](#). See Section [\[\[#anchor:extendable\]\[2.3 \[extendable\], page \]\]](#). See Section [\[\[#anchor:mutable\]\[2.4 \[mutable\], page \]\]](#). See Section [\[\[#anchor:enumerable\]\[2.6 \[enumerable\], page \]\]](#).

plus 1 fill `=hash-bag? =/object=/` [procedure]

returns a boolean indicating whether or not `object` is a hash-bag.

plus 1 fill `make-hash-bag =/[comparator]=/` [procedure]

requires `:comparator` be an object implementing the `comparator` protocol. `comparator` must be hashable. If it is not provided the default hashtable equality and hash functions are used.

returns an instance of `hash-bag`

plus 1 fill `hash-bag =/[comparator] . items=/` [procedure]

requires `:comparator` be an object implementing the `comparator` protocol and `items` be a list of objects for which `comparator` is applicable. `comparator` must be hashable. If it is not provided the default hashtable equality and hash functions are used.

returns an instance of `hash-bag`

modifies the returned `hash-bag` so that it contains `items`

plus 1 fill `=hash-bag-copy =/bag=/` [procedure]

requires `bag` be a hash-bag

returns a shallow copy of `bag`.

plus 1 fill `=hash-bag-empty? =/bag=/` [procedure]

requires `bag` be a hash-bag

returns a boolean indicating whether or not `bag` contains any items.

plus 1 fill =hash-bag-insert! =/=bag item=/ [procedure]

requires **bag** be a hash-bag, and **item** be an arbitrary object supported by the comparator used to create **bag**.

modifies **bag** so that it contains a copy or an additional copy, if one already exists, of **item**.

returns unspecified

plus 1 fill =hash-bag-delete! =/=bag item=/ [procedure]

requires **bag** be a hash-bag, and **item** be an arbitrary object supported by the comparator used to create **bag**.

modifies **bag** so that it contains one less copy of **item**. If 0 copies of **item** exist it is removed from **bag**.

returns unspecified

plus 1 fill =hash-bag-contains? =/=bag item=/ [procedure]

requires **bag** be a hash-bag, and **item** be an arbitrary object supported by the comparator used to create **bag**

returns a boolean indicating whether or not the **bag** contains **item**.

plus 1 fill =hash-bag-count =/=bag item=/ [procedure]

requires **bag** be a hash-bag, and **item** be an arbitrary object supported by the comparator used to create **bag**

returns the number of **item**s found in **bag**.

plus 1 fill =hash-bag-count-set! =/=bag item count=/ [procedure]

requires **bag** be a hash-bag, **item** be an arbitrary object supported by the comparator used to create **bag**, and **count**

be an integer representing the number of **items** to include in **=bag**

modifies **bag** to contain **count** number of **items** unless **count** is less than or equal to 0 which results in the all **items** begin removed from **=bag**.

returns **#unspecified**

plus 1 fill **=bag-length** **=/bag=/** [procedure]

requires **bag** be a hash-bag

returns the number of items contained in **bag**

3.4.3 Examples

2 procedures are provided for creating a **hash-bag**. The first creates an empty bag and the other populates the bag with the items passed to it:

```
== (enumerable-collect (make-hash-bag :comparator number-comparator)  
list-collector) => ()  
(enumerable-collect (hash-bag :comparator number-comparator 1 1 1 3)  
list-collector)  
=> (1 1 1 3)
```

Assuming a hash-bag has been created as above, to insert an item into a hash-bag use:

```
== (hash-bag-insert! bag 1) => #unspecified
```

The count of an item can be obtained with:

```
== (hash-bag-count bag 1) => 1
```

```
(hash-bag-count bag 2) => 0
```

And the count can be explicitly set using:

```
== (hash-bag-count-set! bag 1 4) => #unspecified
```

```
(hash-bag-count bag 1) => 4
```

Or a single copy deleted with:

```
== (hash-bag-delete! bag 1) => #unspecified
```

```
(hash-bag-count bag 1) => 3
```

To check to see if a hash-bag is empty:

```
== (hash-bag-empty? bag) => #f
```

To check whether an item is a member of a hash-bag:

```
== (hash-bag-contains? bag 1) => #t
```

```
(hash-bag-contains? bag 3) => #f
```

And finally, to make a copy:

```
== (let ((bag2 (hash-bag-copy? bag))) (eq? bag bag2)) => #f
```

hash-bag also implements the **bag**, **collection**, **mutable**, and **enumerable** protocols. See Section [\[\[#anchor:bag-Examples\]\[2.9.3 \[bag Examples\], page \]\]](#). See Section [\[\[#anchor:collection-Examples\]\[2.1.3 \[collection Examples\], page \]\]](#). See Section [\[\[#anchor:extendable-Examples\]\[2.3.3 \[extendable Examples\], page \]\]](#). See Section [\[\[#anchor:mutable-Examples\]\[2.4.3 \[mutable Examples\], page \]\]](#). See Section [\[\[#anchor:enumerable-Examples\]\[2.6.3 \[enumerable Examples\], page \]\]](#).

3.5 sorted-set

3.5.1 Overview

sorted-set is an implementation of **set**. It is based on a balanced-binary tree and keeps the elements of the set sorted.

3.5.2 API Reference

sorted-set implements the **set**, **collection**, **extendable**, **mutable**, and **enumerable** protocols. See Section [\[\[#anchor:set\]\[2.10 \[set\], page \]\]](#). See Section [\[\[#anchor:collection\]\[2.1 \[collection\], page \]\]](#). See Section [\[\[#anchor:extendable\]\[2.3 \[extendable\], page \]\]](#). See Section [\[\[#anchor:mutable\]\[2.4 \[mutable\], page \]\]](#). See Section [\[\[#anchor:enumerable\]\[2.6 \[enumerable\], page \]\]](#).

```
plus 1 fill =sorted-set? =/object=/ [procedure]
```

returns a boolean indicating whether or not **object** is a sorted-set.

```
plus 1 fill make-sorted-set =/:comparator=/ [procedure]
```

requires **:comparator** be an object implementing the **comparator** protocol. **comparator** must be ordered.

returns an instance of **sorted-set**

```
plus 1 fill sorted-set =/:comparator . items=/ [procedure]
```

requires : `comparator` be an object implementing the `comparator` protocol and `items` be a list of objects for which `comparator` is applicable. `comparator` must be ordered.

returns an instance of `sorted-set`

modifies the returned `sorted-set` so that it contains `items`

plus 1 fill `=sorted-set-copy /=set=/` [procedure]

requires `set` be a sorted-set

returns a shallow copy of `set`.

plus 1 fill `=sorted-set-empty? /=set=/` [procedure]

requires `set` be a sorted-set

returns a boolean indicating whether or not `set` contains any items.

plus 1 fill `=sorted-set-insert! /=set item=/` [procedure]

requires `set` be a sorted-set, and `item` be an arbitrary object supported by the comparator used to create `set`.

modifies `set` so that it contains a copy or an additional copy, if one already exists, of `item`.

returns unspecified

plus 1 fill `=sorted-set-delete! /=set item=/` [procedure]

requires `set` be a sorted-set, and `item` be an arbitrary object supported by the comparator used to create `set`.

modifies `set` so that it contains one less copy of `item`. If 0 copies of `item` exist it is removed from `set`.

returns unspecified

plus 1 fill =sorted-set-contains? =/=set item=/ [procedure]

requires **set** be a sorted-set, and **item** be an arbitrary object supported by the comparator used to create **set**

returns a boolean indicating whether or not the **set** contains **item**.

plus 1 fill =set-length =/=set=/ [procedure]

requires **set** be a sorted-set

returns the number of items contained in **set**

3.5.3 Examples

2 procedures are provided for creating a **sorted-set**. The first creates an empty set and the other populates the set with the items passed to it:

```
== (enumerable-collect (make-sorted-set :comparator number-comparator)  
list-collector) => ()  
(enumerable-collect (sorted-set :comparator number-comparator 1 1 1 3)  
list-collector)  
=> (1 3)
```

Assuming a sorted-set has been created as above, to insert an item into a sorted-set use:

```
== (sorted-set-insert! set 1) => #unspecified
```

Or deleted with:

```
== (sorted-set-delete! set 1) => #unspecified
```

To check to see if a sorted-set is empty:

```
== (sorted-set-empty? set) => #f
```

To check whether an item is a member of a sorted-set:

```
== (sorted-set-contains? set 1) => #t
```

```
(sorted-set-contains? set 3) => #f
```

And finally, to make a copy:

```
== (let ((set2 (sorted-set-copy? set))) (eq? set set2)) => #f
```

sorted-set also implements the **set**, **collection**, **mutable**, and **enumerable** protocols. See Section [\[\[#anchor:set-Examples\]\[2.10.3 \[set Examples\], page \]\]](#). See Section [\[\[#anchor:collection-Examples\]\[2.1.3 \[collection Examples\], page \]\]](#). See Section [\[\[#anchor:extendable-Examples\]\[2.3.3 \[extendable Examples\], page \]\]](#) See Section [\[\[#anchor:mutable-Examples\]\[2.4.3 \[mutable Examples\],](#)

page [\]\]](#). See Section [\[\[#anchor:enumerable-Examples\]\[2.6.3 \[enumerable Examples\], page \[\\]\\]\]\(#\)](#).

3.6 hash-set

3.6.1 Overview

hash-set is an implementation of set. As its name would imply, it is a hashtable-based implementation.

3.6.2 API Reference

hash-set implements the **set**, **collection**, **extendable**, **mutable**, and **enumerable** protocols. See Section [\[\[#anchor:set\]\[2.10 \[set\], page \[\\]\\]\]\(#\)](#). See Section [\[\[#anchor:collection\]\[2.1 \[collection\], page \[\\]\\]\]\(#\)](#). See Section [\[\[#anchor:extendable\]\[2.3 \[extendable\], page \[\\]\\]\]\(#\)](#). See Section [\[\[#anchor:mutable\]\[2.4 \[mutable\], page \[\\]\\]\]\(#\)](#). See Section [\[\[#anchor:enumerable\]\[2.6 \[enumerable\], page \[\\]\\]\]\(#\)](#).

plus 1 fill **=hash-set? =/object=/** [procedure]

returns a boolean indicating whether or not **object** is a hash-set.

plus 1 fill **make-hash-set =/[comparator]=/** [procedure]

requires **:comparator** be an object implementing the **comparator** protocol. **comparator** must be hashable. If it is not provided the default hashtable equality and hash functions are used.

returns an instance of **hash-set**

plus 1 fill **hash-set =/[comparator] . items=/** [procedure]

requires **:comparator** be an object implementing the **comparator** protocol and **items** be a list of objects for which **comparator** is applicable. **comparator** must be hashable. If it is not provided the default hashtable equality and hash functions are used.

returns an instance of **hash-set**

modifies the returned **hash-set** so that it contains **items**

plus 1 fill =hash-set-copy /=set=/ [procedure]

requires **set** be a hash-set

returns a shallow copy of **set**.

plus 1 fill =hash-set-empty? /=set=/ [procedure]

requires **set** be a hash-set

returns a boolean indicating whether or not **set** contains any items.

plus 1 fill =hash-set-insert! /=set item=/ [procedure]

requires **set** be a hash-set, and **item** be an arbitrary object supported by the comparator used to create **set**.

modifies **set** so that it contains a copy or an additional copy, if one already exists, of **item**.

returns unspecified

plus 1 fill =hash-set-delete! /=set item=/ [procedure]

requires **set** be a hash-set, and **item** be an arbitrary object supported by the comparator used to create **set**.

modifies **set** so that it contains one less copy of **item**. If 0 copies of **item** exist it is removed from **set**.

returns unspecified

plus 1 fill =hash-set-contains? /=set item=/ [procedure]

requires **set** be a hash-set, and **item** be an arbitrary object supported by the comparator used to create **set**

returns a boolean indicating whether or not the **set** contains **item**.

plus 1 fill =set-length /=set= [procedure]

requires `set` be a hash-set

returns the number of items contained in `set`

3.6.3 Examples

2 procedures are provided for creating a **hash-set**. The first creates an empty set and the other populates the set with the items passed to it:

```
== (enumerable-collect (make-hash-set :comparator number-comparator)  
list-collector) => ()  
(enumerable-collect (hash-set :comparator number-comparator 1 1 1 3)  
list-collector)  
=> (1 3)
```

Assuming a hash-set has been created as above, to insert an item into a hash-set use:

```
== (hash-set-insert! set 1) => #unspecified
```

Or deleted with:

```
== (hash-set-delete! set 1) => #unspecified
```

To check to see if a hash-set is empty:

```
== (hash-set-empty? set) => #f
```

To check whether an item is a member of a hash-set:

```
== (hash-set-contains? set 1) => #t
```

```
(hash-set-contains? set 3) => #f
```

And finally, to make a copy:

```
== (let ((set2 (hash-set-copy? set))) (eq? set set2)) => #f
```

hash-set also implements the **set**, **collection**, **mutable**, and **enumerable** protocols. See Section [\[\[#anchor:set-Examples\]\[2.10.3 \[set Examples\], page \]\]](#). See Section [\[\[#anchor:collection-Examples\]\[2.1.3 \[collection Examples\], page \]\]](#). See Section [\[\[#anchor:extendable-Examples\]\[2.3.3 \[extendable Examples\], page \]\]](#) See Section [\[\[#anchor:mutable-Examples\]\[2.4.3 \[mutable Examples\], page \]\]](#). See Section [\[\[#anchor:enumerable-Examples\]\[2.6.3 \[enumerable Examples\], page \]\]](#).

3.7 stretchy-vector

3.7.1 Overview

stretchy-vector is an auto-resizing vector; it grows and shrinks as items are added and deleted. It provides amortized constant time access.

3.7.2 API Reference

`stretchy-vector` implements the `collection`, `extendable`, `indexable`, `mutable`, and `enumerable` protocols. See Section [\[\[#anchor:set\]\[2.10 \[set\], page \]\]](#). See Section [\[\[#anchor:collection\]\[2.1 \[collection\], page \]\]](#). See Section [\[\[#anchor:extendable\]\[2.3 \[extendable\], page \]\]](#). See Section [\[\[#anchor:indexable\]\[2.2 \[indexable\], page \]\]](#). See Section [\[\[#anchor:mutable\]\[2.4 \[mutable\], page \]\]](#). See Section [\[\[#anchor:enumerable\]\[2.6 \[enumerable\], page \]\]](#).

`plus 1 fill =stretchy-vector? =/=object=/` [procedure]

returns a boolean indicating whether or not `object` is a stretchy-vector.

`plus 1 fill =make-stretchy-vector =/=len [:capacity len] [fill]=/` [procedure]

requires `len` be an integer specifying the requested initial vector length, `:capacity` the initial capacity, and `fill` be an arbitrary value to initialize each element. If not provided, `fill` defaults to `#unspecified` and `:capacity` is equal to `len`, unless `len` is less than the minimal stretchy-vector capacity then it is the minimal stretchy-vector capacity (i.e., 16).

returns a stretchy-vector with an initial capacity of at least `len`.

`plus 1 fill stretchy-vector =/. elems=/` [procedure]

requires `elems` be a list of arbitrary objects.

returns a stretchy-vector with initial elements `elems`.

`plus 1 fill =stretchy-vector-length =/=vec=/` [procedure]

requires `vec` be a stretchy-vector.

returns the length of the stretchy-vector `vec`.

`plus 1 fill =stretchy-vector-expand! =/=vec len=/` [procedure]

requires `vec` be a stretchy-vector and `len` be a positive integer.

returns a `vec` expanded to at least `len`.

plus 1 fill `=stretchy-vector-resize! =/=vec new-len=/` [procedure]

requires `vec` be a stretchy-vector and `new-len` be a positive integer.

modifies `vec` such that if `new-len` is larger than the the current length then all of the new elements are set to `#unspecified`, and if `new-len` is smaller than the current length, the elements greater than or equal to `new-len` are dropped.

returns a `vec` resized to `new-len`.

plus 1 fill `=stretchy-vector-capacity =/=vec=/` [procedure]

requires `vec` be a stretchy-vector.

returns the capacity of the stretchy-vector `vec`.

plus 1 fill `=stretchy-vector-set! =/=vec index val=/` [procedure]

requires `vec` be a stretchy-vector, `index` be a positive integer, and `val` be an arbitrary object.

modifies `vec` so that the value at `index` is `value`. This may require expanding the `vec`, if it currently does not contain `index`.

returns `#unspecified`.

plus 1 fill `=stretchy-vector-ref =/=vec index=/` [procedure]

requires `vec` be a stretchy-vector and `index` be a positive integer.

returns the value found at `index` or throws `&invalid-index-exception` if `index` is not in `vec`.

plus 1 fill =list->stretchy-vector /=lst= [procedure]

requires `lst` be a list of arbitrary objects.

returns a stretchy-vector containing the elements of `lst`.

plus 1 fill =stretchy-vector->list /=vec= [procedure]

requires `vec` be a stretchy-vector of arbitrary objects.

returns a list containing the elements of `vec`.

plus 1 fill =vector->stretchy-vector /=vec= [procedure]

requires `vec` be a vector of arbitrary objects.

returns a stretchy-vector containing the elements of `vec`.

plus 1 fill =stretchy-vector->vector /=vec= [procedure]

requires `vec` be a stretchy-vector of arbitrary objects.

returns a vector containing the elements of `vec`.

plus 1 fill =stretchy-vector-map /=proc vec= [procedure]

requires `proc` be a single argument procedure compatible with being called element-wise to `vec` and `vec` be a stretchy-vector of arbitrary objects.

returns a new stretchy-vector containing the results of applying `proc` to the elements of `vec`.

plus 1 fill =stretchy-vector-map /=proc vec= [procedure]

requires `proc` be a single argument procedure compatible with being called element-wise to `vec` and `vec` be a stretchy-vector of arbitrary objects.

modifies the elements of `vec` so that they are the value obtained by applying `proc` to each element.

returns `vec`.

plus 1 fill =stretchy-vector-copy /=vec=/ [procedure]

requires `vec` be a stretchy-vector of arbitrary objects.

returns a shallow copy of `vec`.

plus 1 fill =stretchy-vector-extend! /=vec val=/ [procedure]

requires `vec` be a stretchy-vector of arbitrary objects and `val` be an arbitrary object.

modifies `vect` by adding `val` to the end.

returns #unspecified.

plus 1 fill =stretchy-vector-remove! /=vec=/ [procedure]

requires `vec` be a stretchy-vector of arbitrary objects

modifies `vect` by removing the last element.

returns the element removed from `vec`.

plus 1 fill =stretchy-vector-append /=vec1 vec2=/ [procedure]

requires `vec1` and `vec2` be stretchy-vectors of arbitrary objects

returns a new stretchy-vector containing the elements of `vec1` followed by those in `vec2`.

plus 1 fill =stretchy-vector-append! /=vec1 vec2=/ [procedure]

requires `vec1` and `vec2` be stretchy-vectors of arbitrary objects

modifies `vec1` so that its current elements are followed by the elements of `vec2`. **returns** #unspecified.

3.7.3 Examples

To test whether an object is a **stretchy-vector** use the predicate **stretchy-vector?**:

```
== (stretchy-vector? (stretchy-vector)) ⇒ #t
(stretchy-vector? (vector)) ⇒ #f
```

Two procedures are used to create **=stretchy-vector=s**. The first creates an empty vector of a specified size with an optional fill value, and the second allows for the creation of a stretchy vector containing the passed values.

```
== (let ((vec (make-stretchy-vector 3 9))) (enumerable-collect vec list-collector))
⇒ (9 9 9)
(let ((vec (stretchy-vector 1 2 3))) (enumerable-collect vec list-collector))
⇒ (1 2 3)
```

As with regular vectors, you can reference and set the values of individual **stretchy-vector** elements:

```
== (let ((vec (stretchy-vector 1 2 3))) (stretchy-vector-ref vec 1)) ⇒ 2
(let ((vec (stretchy-vector 4 5 6))) (stretchy-vector-set! vec 2 7) (stretchy-
vector-ref vec 2)) ⇒ 7
```

The length of a **stretchy-vector** is determined with **stretchy-vector-length**:

```
== (stretchy-vector-length (stretchy-vector 1 2 3 4 5)) ⇒ 5
```

A shallow copy of a **stretchy-vector** is obtained with **stretchy-vector-copy**:

```
== (let* ((vec1 (stretchy-vector 1 2 3)) (vec2 (stretchy-vector-copy
vec1))) (eq? vec1 vec2) ⇒ #f (equal? vec2 vec2) ⇒ #t
```

stretchy-vector also implements the **collection**, **mutable**, **indexable**, **extendable**, **enumerable**, and **dictionary-enumerable** protocols. See Section [\[\[#anchor:collection-Examples\]\[2.1.3 \[collection Examples\], page \]\]](#). See Section [\[\[#anchor:mutable-Examples\]\[2.4.3 \[mutable Examples\], page \]\]](#). See Section [\[\[#anchor:indexable-Examples\]\[2.2.3 \[indexable Examples\], page \]\]](#). See Section [\[\[#anchor:extendable-Examples\]\[2.3.3 \[extendable Examples\], page \]\]](#). See Section [\[\[#anchor:enumerable-Examples\]\[2.6.3 \[enumerable Examples\], page \]\]](#). See Section [\[\[#anchor:dictionary-enumerable-Examples\]\[2.17.3 \[dictionary-enumerable Examples\], page \]\]](#).

3.8 contiguous-stack

3.8.1 Overview

contiguous-stack is a contiguous implementation of a stack, a last-in-first-out data structure, with a finite capacity.

3.8.2 API Reference

`contiguous-stack` implements the `stack`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section [\[\[#anchor:stack\]\[2.13 \[stack\], page \]\]](#). See Section [\[\[#anchor:collection\]\[2.1 \[collection\], page \]\]](#). See Section [\[\[#anchor:extendable\]\[2.3 \[extendable\], page \]\]](#). See Section [\[\[#anchor:mutable\]\[2.4 \[mutable\], page \]\]](#). See Section [\[\[#anchor:enumerable\]\[2.6 \[enumerable\], page \]\]](#).

plus 1 fill `=contiguous-stack? =/=object=/` [procedure]

returns a boolean indicating whether or not `object` is a contiguous-stack.

plus 1 fill `make-contiguous-stack =/:capacity=/` [procedure]

requires `:capacity` be a positive integer specifying the maximum capacity of the stack.

returns a new `contiguous-stack` with a capacity of `:capacity`

plus 1 fill `contiguous-stack =/:capacity . items=/` [procedure]

requires `:capacity` be a positive integer specifying the maximum capacity of the stack and `items` be a list of items to initialize the stack with.

returns a new `contiguous-stack` with a capacity of `:capacity` containing `items` pushed from right-to-left onto the stack. If the number of `items` is greater than `:capacity`, `&invalid-argument-exception` is thrown.

plus 1 fill `=contiguous-stack-copy =/=stack=/` [procedure]

requires `stack` be a `contiguous-stack`.

returns a shallow copy of `stack`.

plus 1 fill `=contiguous-stack-empty? =/=stack=/` [procedure]

requires `stack` be a `contiguous-stack`.

returns a boolean indicating whether or not the stack is empty.

plus 1 fill =contiguous-stack-length /=stack=/ [procedure]

requires `stack` be a `contiguous-stack`.

returns the number of items on the stack (i.e., `length`) .

plus 1 fill =contiguous-stack-capacity /=stack=/ [procedure]

requires `stack` be a `contiguous-stack`.

returns the maximum size (i.e., `length`) of `stack`.

plus 1 fill =contiguous-stack-push! /=stack item=/ [procedure]

requires `stack` be a `contiguous-stack` and `item` be an arbitrary object.

modifies `stack` by pushing `item` on the top of the stack.

returns `#unspecified` or if there is no free capacity, throws `&invalid-state-exception`.

plus 1 fill =contiguous-stack-pop! /=stack=/ [procedure]

requires `stack` be a `contiguous-stack`.

modifies `stack` by removing the item on the top of the stack.

returns the item removed from `stack` or if `stack` is empty, throws `&invalid-state-exception`.

plus 1 fill =contiguous-stack-top /=stack=/ [procedure]

requires `stack` be a `contiguous-stack`.

returns the top item from `stack` or if `stack` is empty, throws `&invalid-state-exception`.

3.8.3 Examples

2 procedures are provided for creating a **contiguous-stack**: The first creates an empty stack and the other populates the stack with the items passed to it:

```
== (enumerable-collect (make-contiguous-stack :capacity 4) list-collector)
⇒ ()
(enumerable-collect (contiguous-stack :capacity 4 1 2 3) list-collector)
⇒ (1 2 3)
```

An item can be pushed onto the stack with **contiguous-stack-push!**:

```
== (let ((stack (contiguous-stack :capacity 4))) (contiguous-stack-push! stack
1) (contiguous-stack-top stack)) ⇒ 1
```

The top item of the stack can be non-destructively obtained with **contiguous-stack-top**:

```
== (let ((stack (contiguous-stack :capacity 4 1 2 3))) (contiguous-stack-top
stack)) ⇒ 1
```

The top item can be removed from the stack with **contiguous-stack-pop!**:

```
== (let ((stack (contiguous-stack :capacity 4 1 2 3))) (contiguous-stack-pop!
stack) (contiguous-stack-top stack)) ⇒ 2
```

To test if a stack is empty, use **contiguous-stack-empty?**:

```
== (contiguous-stack-empty? (contiguous-stack :capacity 4)) ⇒ #t
(contiguous-stack-empty? (contiguous-stack :capacity 4 1 2)) ⇒ #f
```

The size or length of a stack is obtained with **contiguous-stack-length**:

```
== (contiguous-stack-length (contiguous-stack :capacity 4 1 2)) ⇒ 2
```

The capacity or maximum length of a stack is obtained with **contiguous-stack-capacity**:

```
== (contiguous-stack-capacity (contiguous-stack-capacity :capacity 4 1
2)) ⇒ 4
```

To make a shallow copy a stack, use **contiguous-stack-copy**:

```
== (let* ((stack1 (contiguous-stack :capacity 4 1 2)) (stack2 (contiguous-
stack-copy stack1))) (eq? stack1 stack2)) ⇒ #f
```

contiguous-stack also implements the **stack**, **collection**, **extendable**, **mutable**, and **enumerable** protocols. See Section [\[\[#anchor:stack-Examples\]\[2.13.3 \[stack Examples\], page \]\]](#). See Section [\[\[#anchor:collection-Examples\]\[2.1.3 \[collection Examples\], page \]\]](#). See Section [\[\[#anchor:extendable-Examples\]\[2.3.3 \[extendable Examples\], page \]\]](#). See Section [\[\[#anchor:mutable-Examples\]\[2.4.3 \[mutable Examples\], page \]\]](#). See Section [\[\[#anchor:enumerable-Examples\]\[2.6.3 \[enumerable Examples\], page \]\]](#).

3.9 linked-stack

3.9.1 Overview

`linked-stack` is a linked-list based implementation of a stack, a last-in-first-out data structure.

3.9.2 API Reference

`linked-stack` implements the `stack`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section [\[\[#anchor:stack\]\[2.13 \[stack\], page \]\]](#). See Section [\[\[#anchor:collection\]\[2.1 \[collection\], page \]\]](#). See Section [\[\[#anchor:extendable\]\[2.3 \[extendable\], page \]\]](#). See Section [\[\[#anchor:mutable\]\[2.4 \[mutable\], page \]\]](#). See Section [\[\[#anchor:enumerable\]\[2.6 \[enumerable\], page \]\]](#).

plus 1 fill `=linked-stack? =/=object=/` [procedure]

returns a boolean indicating whether or not `object` is a linked-stack.

plus 1 fill `make-linked-stack` [procedure]

returns a new empty `linked-stack`.

plus 1 fill `linked-stack =/. items=/` [procedure]

requires `items` be a list of items to initialize the stack with.

returns a new `linked-stack` containing `items` pushed from right-to-left onto the stack.

plus 1 fill `=linked-stack-copy =/=stack=/` [procedure]

requires `stack` be a `linked-stack`.

returns a shallow copy of `stack`.

plus 1 fill `=linked-stack-empty? =/=stack=/` [procedure]

requires `stack` be a `linked-stack`.

returns a boolean indicating whether or not the stack is empty.

plus 1 fill =linked-stack-length /=stack=/ [procedure]

requires stack be a **linked-stack**.

returns the number of items on the stack (i.e., length) .

plus 1 fill =linked-stack-push! /=stack item=/ [procedure]

requires stack be a **linked-stack** and item be an arbitrary object.

modifies stack by pushing item on the top of the stack.

returns #unspecified.

plus 1 fill =linked-stack-pop! /=stack=/ [procedure]

requires stack be a **linked-stack**.

modifies stack by removing the item on the top of the stack.

returns the item removed from stack or if stack is empty, throws &invalid-state-exception.

plus 1 fill =linked-stack-top /=stack=/ [procedure]

requires stack be a **linked-stack**.

returns the top item from stack or if stack is empty, throws &invalid-state-exception.

3.9.3 Examples

2 procedures are provided for creating a **linked-stack**: The first creates an empty stack and the other populates the stack with the items passed to it:

== (enumerable-collect (make-linked-stack) list-collector) ⇒ ()

(enumerable-collect (linked-stack 1 2 3) list-collector) ⇒ (1 2 3)

An item can be pushed onto the stack with **linked-stack-push!**:

== (let ((stack (linked-stack))) (linked-stack-push! stack 1) (linked-stack-top stack)) ⇒ 1

The top item of the stack can be non-destructively obtained with `linked-stack-top`:

```
== (let ((stack (linked-stack 1 2 3))) (linked-stack-top stack)) => 1
```

The top item can be removed from the stack with `linked-stack-pop!`:

```
== (let ((stack (linked-stack 1 2 3))) (linked-stack-pop! stack) (linked-stack-top stack)) => 2
```

To test if a stack is empty, use `linked-stack-empty?`:

```
== (linked-stack-empty? (linked-stack)) => #t
```

```
(linked-stack-empty? (linked-stack 1 2)) => #f
```

The size or length of a stack is obtained with `linked-stack-length`:

```
== (linked-stack-length (linked-stack 1 2)) => 2
```

To make a shallow copy a stack, use `linked-stack-copy`:

```
== (let* ((stack1 (linked-stack 1 2)) (stack2 (linked-stack-copy stack1)))  
(eq? stack1 stack2)) => #f
```

`linked-stack` also implements the `stack`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section [\[\[#anchor:stack-Examples\]\[2.13.3 \[stack Examples\], page \]\]](#). See Section [\[\[#anchor:collection-Examples\]\[2.1.3 \[collection Examples\], page \]\]](#). See Section [\[\[#anchor:extendable-Examples\]\[2.3.3 \[extendable Examples\], page \]\]](#). See Section [\[\[#anchor:mutable-Examples\]\[2.4.3 \[mutable Examples\], page \]\]](#). See Section [\[\[#anchor:enumerable-Examples\]\[2.6.3 \[enumerable Examples\], page \]\]](#).

3.10 contiguous-queue

3.10.1 Overview

`contiguous-queue` is a contiguous implementation of a queue, a last-in-first-out data structure, with a finite capacity.

3.10.2 API Reference

`contiguous-queue` implements the `queue`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section [\[\[#anchor:queue\]\[2.11 \[queue\], page \]\]](#). See Section [\[\[#anchor:collection\]\[2.1 \[collection\], page \]\]](#). See Section [\[\[#anchor:extendable\]\[2.3 \[extendable\], page \]\]](#). See Section [\[\[#anchor:mutable\]\[2.4 \[mutable\], page \]\]](#). See Section [\[\[#anchor:enumerable\]\[2.6 \[enumerable\], page \]\]](#).

plus 1 fill =contiguous-queue? =/=object=/ [procedure]

returns a boolean indicating whether or not `object` is a contiguous-queue.

plus 1 fill **make-contiguous-queue** *=/:capacity=/* [procedure]

requires *:capacity* be a positive integer specifying the maximum capacity of the queue.

returns a new **contiguous-queue** with a capacity of *:capacity*

plus 1 fill **contiguous-queue** *=/:capacity . items=/* [procedure]

requires *:capacity* be a positive integer specifying the maximum capacity of the queue and *items* be a list of items to initialize the queue with.

returns a new **contiguous-queue** with a capacity of *:capacity* containing *items* enqueued from left-to-right onto the queue. If the number of *items* is greater than *:capacity*, **&invalid-argument-exeption** is thrown.

plus 1 fill **=contiguous-queue-copy** *=/=queue=/* [procedure]

requires *queue* be a **contiguous-queue**.

returns a shallow copy of *queue*.

plus 1 fill **=contiguous-queue-empty?** *=/=queue=/* [procedure]

requires *queue* be a **contiguous-queue**.

returns a boolean indicating whether or not the queue is empty.

plus 1 fill **=contiguous-queue-length** *=/=queue=/* [procedure]

requires *queue* be a **contiguous-queue**.

returns the length the queue.

plus 1 fill **=contiguous-queue-capacity** *=/=queue=/* [procedure]

requires *queue* be a **contiguous-queue**.

returns the maximum size length of *queue*.

plus 1 fill =contiguous-queue-enqueue! =/=queue item=/ [procedure]

requires `queue` be a `contiguous-queue` and `item` be an arbitrary object.

modifies `queue` by placing `item` on the end of the queue.

returns `#unspecified` or if there is no free capacity, throws `&invalid-state-exception`.

plus 1 fill =contiguous-queue-dequeue! =/=queue=/ [procedure]

requires `queue` be a `contiguous-queue`.

modifies `queue` by removing the first item from the queue.

returns the item removed from `queue` or if `queue` is empty, throws `&invalid-state-exception`.

plus 1 fill =contiguous-queue-first =/=queue=/ [procedure]

requires `queue` be a `contiguous-queue`.

returns the first item from `queue` or if `queue` is empty, throws `&invalid-state-exception`.

3.10.3 Examples

2 procedures are provided for creating a `contiguous-queue`: The first creates an empty queue and the other populates the queue with the items passed to it:

```
== (enumerable-collect (make-contiguous-queue :capacity 4) list-collector)
⇒ ()
(enumerable-collect (contiguous-queue :capacity 4 1 2 3) list-collector)
⇒ (1 2 3)
```

An item can be placed onto the queue with `contiguous-queue-enqueue!`:

```
== (let ((queue (contiguous-queue :capacity 4 1 2 3))) (contiguous-queue-enqueue! queue 4) (contiguous-queue-first queue)) ⇒ 1
```

The first item of the queue can be non-destructively obtained with `contiguous-queue-first`:

```
== (let ((queue (contiguous-queue :capacity 4 1 2 3))) (contiguous-queue-first queue)) => 1
```

The first item can be removed from the queue with `contiguous-queue-dequeue!`:

```
== (let ((queue (contiguous-queue :capacity 4 1 2 3))) (contiguous-queue-dequeue! queue) (contiguous-queue-first queue)) => 2
```

To test if a queue is empty, use `contiguous-queue-empty?`:

```
== (contiguous-queue-empty? (contiguous-queue :capacity 4)) => #t  
(contiguous-queue-empty? (contiguous-queue :capacity 4 1 2)) => #f
```

The size or length of a queue is obtained with `contiguous-queue-length`:

```
== (contiguous-queue-length (contiguous-queue :capacity 4 1 2)) => 2
```

The capacity or maximum length of a queue is obtained with `contiguous-queue-capacity`:

```
== (contiguous-queue-capacity (contiguous-queue-capacity :capacity 4 1 2)) => 4
```

To make a shallow copy a queue, use `contiguous-queue-copy`:

```
== (let* ((queue1 (contiguous-queue :capacity 4 1 2)) (queue2 (contiguous-queue-copy queue1))) (eq? queue1 queue2)) => #f
```

`contiguous-queue` also implements the `queue`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section [\[\[#anchor:queue-Examples\]\[2.11.3 \[queue Examples\], page \]\]](#). See Section [\[\[#anchor:collection-Examples\]\[2.1.3 \[collection Examples\], page \]\]](#). See Section [\[\[#anchor:extendable-Examples\]\[2.3.3 \[extendable Examples\], page \]\]](#). See Section [\[\[#anchor:mutable-Examples\]\[2.4.3 \[mutable Examples\], page \]\]](#). See Section [\[\[#anchor:enumerable-Examples\]\[2.6.3 \[enumerable Examples\], page \]\]](#).

3.11 linked-queue

3.11.1 Overview

`linked-queue` is a linked-list based implementation of a queue, a last-in-first-out data structure.

3.11.2 API Reference

`linked-queue` implements the `queue`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section [\[\[#anchor:queue\]\[2.11 \[queue\], page \]\]](#). See Section [\[\[#anchor:collection\]\[2.1 \[collection\], page \]\]](#). See Section [\[\[#anchor:extendable\]\[2.3 \[extendable\], page \]\]](#). See Section [\[\[#anchor:mutable\]\[2.4 \[mutable\], page \]\]](#). See Section [\[\[#anchor:enumerable\]\[2.6 \[enumerable\], page \]\]](#).

plus 1 fill `=linked-queue? =/=object=` [procedure]

returns a boolean indicating whether or not `object` is a linked-queue.

plus 1 fill `make-linked-queue` [procedure]

returns a new `linked-queue`.

plus 1 fill `linked-queue =/. items=` [procedure]

requires `items` be a list of items to initialize the queue with.

returns a new `linked-queue` containing `items` enqueued from left-to-right onto the queue.

plus 1 fill `=linked-queue-copy =/=queue=` [procedure]

requires `queue` be a `linked-queue`.

returns returns a shallow copy of `queue`.

plus 1 fill `=linked-queue-empty? =/=queue=` [procedure]

requires `queue` be a `linked-queue`.

returns returns a boolean indicating whether or not the queue is empty.

plus 1 fill `=linked-queue-length =/=queue=` [procedure]

requires `queue` be a `linked-queue`.

returns returns the length the queue.

plus 1 fill `=linked-queue-enqueue! =/=queue item=` [procedure]

requires `queue` be a `linked-queue` and `item` be an arbitrary object.

modifies `queue` by placing `item` on the end of the queue.

returns `#unspecified`.

plus 1 fill =linked-queue-dequeue! /=queue=/ [procedure]

requires `queue` be a `linked-queue`.

modifies `queue` by removing the first item from the queue.

returns the item removed from `queue` or if `queue` is empty, throws `&invalid-state-exception`.

plus 1 fill =linked-queue-first /=queue=/ [procedure]

requires `queue` be a `linked-queue`.

returns the first item from `queue` or if `queue` is empty, throws `&invalid-state-exception`.

3.11.3 Examples

2 procedures are provided for creating a `linked-queue`: The first creates an empty queue and the other populates the queue with the items passed to it:

```
== (enumerable-collect (make-linked-queue) list-collector) => ()  
(enumerable-collect (linked-queue 1 2 3) list-collector) => (1 2 3)
```

An item can be placed onto the queue with `linked-queue-enqueue!`:

```
== (let ((queue (linked-queue 1 2 3))) (linked-queue-enqueue! queue  
4) (linked-queue-first queue)) => 1
```

The first item of the queue can be non-destructively obtained with `linked-queue-first`:

```
== (let ((queue (linked-queue 1 2 3))) (linked-queue-first queue)) =>  
1
```

The first item can be removed from the queue with `linked-queue-dequeue!`:

```
== (let ((queue (linked-queue 1 2 3))) (linked-queue-dequeue! queue)  
(linked-queue-first queue)) => 2
```

To test if a queue is empty, use `linked-queue-empty?`:

```
== (linked-queue-empty? (linked-queue)) => #t  
(linked-queue-empty? (linked-queue 1 2)) => #f
```

The size or length of a queue is obtained with `linked-queue-length`:

```
== (linked-queue-length (linked-queue 1 2)) => 2
```

To make a shallow copy a queue, use `linked-queue-copy`:

```
== (let* ((queue1 (linked-queue 1 2)) (queue2 (linked-queue-copy queue1)))
  (eq? queue1 queue2)) => #f
```

`linked-queue` also implements the `queue`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section [\[\[#anchor:queue-Examples\]\[2.11.3 \[queue Examples\], page \]\]](#). See Section [\[\[#anchor:collection-Examples\]\[2.1.3 \[collection Examples\], page \]\]](#). See Section [\[\[#anchor:extendable-Examples\]\[2.3.3 \[extendable Examples\], page \]\]](#). See Section [\[\[#anchor:mutable-Examples\]\[2.4.3 \[mutable Examples\], page \]\]](#). See Section [\[\[#anchor:enumerable-Examples\]\[2.6.3 \[enumerable Examples\], page \]\]](#).

3.12 ring-buffer

3.12.1 Overview

`ring-buffer` is a fixed-size, sequential data structure supporting the efficient access, addition and removal of elements from both the front and back. It is commonly used to implement the queue and deque ADTs.

3.12.2 API Reference

`ring-buffer` implements the `queue`, `deque`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section [\[\[#anchor:queue\]\[2.11 \[queue\], page \]\]](#). See Section [\[\[#anchor:deque\]\[2.12 \[deque\], page \]\]](#). See Section [\[\[#anchor:collection\]\[2.1 \[collection\], page \]\]](#). See Section [\[\[#anchor:extendable\]\[2.3 \[extendable\], page \]\]](#). See Section [\[\[#anchor:mutable\]\[2.4 \[mutable\], page \]\]](#). See Section [\[\[#anchor:enumerable\]\[2.6 \[enumerable\], page \]\]](#).

```
plus 1 fill =ring-buffer? =/object=/ [procedure]
```

returns a boolean indicating whether or not `object` is a ring-buffer.

```
plus 1 fill make-ring-buffer =/:capacity=/ [procedure]
```

returns a new `ring-buffer` with a maximum capacity of `:capacity`.

```
plus 1 fill ring-buffer =/:capacity . items=/ [procedure]
```

requires `:capacity` be a positive integer specifying the maximum capacity of the ring-buffer and `items` be a list of items to initialize the ring-buffer with.

returns a new `ring-buffer` with a capacity of `:capacity` containing `items` pushed on the back from left-to-right. If the number of `items` is greater than `:capacity`, `&invalid-argument-exception` is thrown.

plus 1 fill `=ring-buffer-copy =/=rb=` [procedure]

requires `rb` be a `ring-buffer`.

returns a shallow copy of `rb`.

plus 1 fill `=ring-buffer-empty? =/=rb=` [procedure]

requires `rb` be a `ring-buffer`.

returns a boolean indicating whether or not the ring-buffer is empty.

plus 1 fill `=ring-bufer-length =/=rb=` [procedure]

requires `rb` be a `ring-buffer`.

returns the size/length the ring-buffer.

plus 1 fill `=ring-buffer-capacity =/=rb=` [procedure]

requires `rb` be a `ring-buffer`.

returns the maximum size/length of `rb`.

plus 1 fill `=ring-buffer-front =/=rb=` [procedure]

requires `rb` be a `ring-buffer`.

returns the item at the front of the ring buffer or if `rb` is empty, throws `&invalid-state-exception`.

plus 1 fill `=ring-buffer-back =/=rb=` [procedure]

requires `rb` be a `ring-buffer`.

returns the item at the back of the ring buffer or if `rb` is empty, throws `&invalid-state-exception`.

plus 1 fill `=ring-buffer-pop-front! /=rb=/` [procedure]

requires `rb` be a `ring-buffer`.

modifies `rb` by removing the first item from the ring-buffer.

returns the item removed from the front of the ring buffer or if `rb` is empty, throws `&invalid-state-exception`.

plus 1 fill `=ring-buffer-pop-back! /=rb=/` [procedure]

requires `rb` be a `ring-buffer`.

modifies `rb` by removing the back item from the ring-buffer.

returns the item removed from the back of the ring buffer or if `rb` is empty, throws `&invalid-state-exception`.

plus 1 fill `=ring-buffer-push-front! /=rb item=/` [procedure]

requires `rb` be a `ring-buffer` and `item` be an arbitrary object.

modifies `rb` by pushing `item` on the front of the ring-buffer.

returns `#unspecified` or if `rb` is full, throws `&invalid-state-exception`.

plus 1 fill `=ring-buffer-push-back! /=rb item=/` [procedure]

requires `rb` be a `ring-buffer` and `item` be an arbitrary object.

modifies `rb` by pushing `item` on the back of the ring-buffer.

returns `#unspecified` or if `rb` is full, throws `&invalid-state-exception`.

3.12.3 Examples

2 procedures are provided for creating a **ring-buffer**: The first creates an empty ring-buffer and the other populates the ring-buffer with the items passed to it:

```
== (enumerable-collect (make-ring-buffer :capacity 4) list-collector)
⇒ ()
(enumerable-collect (ring-buffer :capacity 4 1 2 3) list-collector) ⇒ (1
2 3)
```

An item can be placed onto the front or back of the ring-buffer with **ring-buffer-push-back!** and **ring-buffer-push-front!**:

```
== (let ((rb (ring-buffer :capacity 4 1 2 3))) (ring-buffer-push-back! rb
4) (ring-buffer-push-front! rb 0) (cons (ring-buffer-back rb) (ring-buffer-
front rb)) ⇒ (4 . 0)
```

The first and last item of the ring-buffer can be non-destructively obtained with **ring-buffer-front** and **ring-buffer-back**, respectively:

```
== (let ((rb (ring-buffer :capacity 4 1 2 3))) (cons (ring-buffer-front rb)
(ring-buffer-back rb)) ⇒ (1 . 3)
```

The first and last items can be removed from the ring-buffer with **ring-buffer-pop-front!** and **ring-buffer-pop-back!**, respectively:

```
== (let ((rb (ring-buffer :capacity 4 1 2 3))) (cons (ring-buffer-pop-
front! rb) (ring-buffer-pop-back! rb)) ⇒ (1 . 3)
```

To test if a ring-buffer is empty, use **ring-buffer-empty?**:

```
== (ring-buffer-empty? (ring-buffer :capacity 4)) ⇒ #t
(ring-buffer-empty? (ring-buffer :capacity 4 1 2)) ⇒ #f
```

The size or length of a ring-buffer is obtained with **ring-buffer-length**:

```
== (ring-buffer-length (ring-buffer :capacity 4 1 2)) ⇒ 2
```

The capacity or maximum length of a ring-buffer is obtained with **ring-buffer-capacity**:

```
== (ring-buffer-capacity (ring-buffer-capacity :capacity 4 1 2)) ⇒ 4
```

To make a shallow copy a ring-buffer, use **ring-buffer-copy**:

```
== (let* ((rb1 (ring-buffer :capacity 4 1 2)) (rb2 (ring-buffer-copy
queue1))) (eq? rb1 rb2)) ⇒ #f
```

ring-buffer also implements the **queue**, **deque**, **collection**, **extendable**, **mutable**, and **enumerable** protocols. See Section [\[\[#anchor:queue-Examples\]\[2.11.3 \[queue Examples\], page \]\]](#). See Section [\[\[#anchor:deque-Examples\]\[2.12.3 \[deque Examples\], page \]\]](#). See Section [\[\[#anchor:collection-Examples\]\[2.1.3 \[collection Examples\], page \]\]](#). See Section [\[\[#anchor:extendable-Examples\]\[2.3.3 \[extendable Examples\], page \]\]](#). See Section [\[\[#anchor:mutable-Examples\]\[2.4.3](#)

[mutable Examples], page]]. See Section [[#anchor:enumerable-Examples][2.6.3 [enumerable Examples], page]].

3.13 linked-deque

3.13.1 Overview

linked-deque is a doubly linked list based implementation of the queue and deque ADTs. It supports efficient access, addition, and removal from both the front and back.

3.13.2 API Reference

linked-deque implements the **queue**, **deque**, **collection**, **extendable**, **mutable**, and **enumerable** protocols. See Section [[#anchor:queue][2.11 [queue], page]]. See Section [[#anchor:deque][2.12 [deque], page]]. See Section [[#anchor:collection][2.1 [collection], page]]. See Section [[#anchor:extendable][2.3 [extendable], page]]. See Section [[#anchor:mutable][2.4 [mutable], page]]. See Section [[#anchor:enumerable][2.6 [enumerable], page]].

plus 1 fill **=linked-deque? =/object=/** [procedure]

returns a boolean indicating whether or not **object** is a linked-deque.

plus 1 fill **make-linked-deque** [procedure]

returns a new **linked-deque**.

plus 1 fill **linked-deque =/. items=/** [procedure]

requires **items** be a list of items to initialize the linked-deque with.

returns a new **linked-deque** containing **items** enqueued from left-to-right.

plus 1 fill **=linked-deque-copy =/lq=/** [procedure]

requires **lq** be a **linked-deque**.

returns a shallow copy of **lq**.

plus 1 fill =linked-deque-empty? =/=lq=/ [procedure]

requires `lq` be a `linked-deque`.

returns a boolean indicating whether or not the linked-deque is empty.

plus 1 fill =ring-bufer-length =/=lq=/ [procedure]

requires `lq` be a `linked-deque`.

returns the size/length the linked-deque.

plus 1 fill =linked-deque-first =/=lq=/ [procedure]

requires `lq` be a `linked-deque`.

returns the first item of the ring buffer or if `lq` is empty, throws `&invalid-state-exception`.

plus 1 fill =linked-deque-last =/=lq=/ [procedure]

requires `lq` be a `linked-deque`.

returns the last item the ring buffer or if `lq` is empty, throws `&invalid-state-exception`.

plus 1 fill =linked-deque-dequeue! =/=lq=/ [procedure]

requires `lq` be a `linked-deque`.

modifies `lq` by removing the first item from the linked-deque.

returns the item removed from the front of the ring buffer or if `lq` is empty, throws `&invalid-state-exception`.

plus 1 fill =linked-deque-dequeue-back! =/=lq=/ [procedure]

requires `lq` be a `linked-deque`.

modifies `lq` by removing the back item from the linked-deque.

returns the item removed from the back of the ring buffer or if `lq` is empty, throws `&invalid-state-exception`.

plus 1 fill `=linked-deque-enqueue-front! =/=lq item=/` [procedure]

requires `lq` be a `linked-deque` and `item` be an arbitrary object.

modifies `lq` by enqueueing `item` on the front of the linked-deque.

returns `#unspecified`.

plus 1 fill `=linked-deque-enqueue! =/=lq item=/` [procedure]

requires `lq` be a `linked-deque` and `item` be an arbitrary object.

modifies `lq` by enqueueing `item` on the back of the linked-deque.

returns `#unspecified`.

3.13.3 Examples

2 procedures are provided for creating a `linked-deque`: The first creates an empty linked-deque and the other populates the linked-deque with the items passed to it:

```
== (enumerable-collect (make-linked-deque) list-collector) => ()  
(enumerable-collect (linked-deque 1 2 3) list-collector) => (1 2 3)
```

An item can be placed onto the front or back of the linked-deque with `linked-deque-enqueue!` and `linked-deque-enqueue-front!`:

```
== (let ((lq (linked-deque 1 2 3))) (linked-deque-enqueue! lq 4) (linked-  
deque-enqueue-front! lq 0) (cons (linked-deque-last lq) (linked-deque-first  
lq)) => (4 . 0)
```

The first and last item of the linked-deque can be non-destructively obtained with `linked-deque-first` and `linked-deque-last`, respectively:


```
== (let ((lq (linked-deque 1 2 3))) (cons (linked-deque-first lq) (linked-deque-last lq))) => (1 . 3)
```

The first and last items can be removed from the linked-deque with `linked-deque-dequeue!` and `linked-deque-dequeue-back!`, respectively:

```
== (let ((lq (linked-deque 1 2 3))) (cons (linked-deque-dequeue! lq) (linked-deque-dequeue-back! lq))) => (1 . 3)
```

To test if a linked-deque is empty, use `linked-deque-empty?`:

```
== (linked-deque-empty? (linked-deque)) => #t
```

```
(linked-deque-empty? (linked-deque 1 2)) => #f
```

The size or length of a linked-deque is obtained with `linked-deque-length`:

```
== (linked-deque-length (linked-deque 1 2)) => 2
```

To make a shallow copy a linked-deque, use `linked-deque-copy`:

```
== (let* ((lq1 (linked-deque 1 2)) (lq2 (linked-deque-copy queue1))) (eq? lq1 lq2)) => #f
```

`linked-deque` also implements the `queue`, `deque`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section [\[\[#anchor:queue-Examples\]\[2.11.3 \[queue Examples\], page \]\]](#). See Section [\[\[#anchor:deque-Examples\]\[2.12.3 \[deque Examples\], page \]\]](#). See Section [\[\[#anchor:collection-Examples\]\[2.1.3 \[collection Examples\], page \]\]](#). See Section [\[\[#anchor:extendable-Examples\]\[2.3.3 \[extendable Examples\], page \]\]](#). See Section [\[\[#anchor:mutable-Examples\]\[2.4.3 \[mutable Examples\], page \]\]](#). See Section [\[\[#anchor:enumerable-Examples\]\[2.6.3 \[enumerable Examples\], page \]\]](#).

3.14 binary-heap

3.14.1 Overview

A `binary-heap` is an implementation of a priority queue featuring contiguous storage and a specified maximum size or capacity.

3.14.2 API Reference

`binary-heap` implements the `priority-queue`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section [\[\[#anchor:priority002dqueue\]\[2.14 \[priority-queue\], page \]\]](#). See Section [\[\[#anchor:collection\]\[2.1 \[collection\], page \]\]](#). See Section [\[\[#anchor:extendable\]\[2.3 \[extendable\], page \]\]](#). See Section [\[\[#anchor:mutable\]\[2.4 \[mutable\], page \]\]](#). See Section [\[\[#anchor:enumerable\]\[2.6 \[enumerable\], page \]\]](#).

```
plus 1 fill =binary-heap? =/=object=/ [procedure]
```

returns a boolean indicating whether or not `object` is a binary-heap.

plus 1 fill `make-binary-heap` `=/:capacity :comparator=/` [procedure]

requires `:capacity` be a positive integer specifying the maximum capacity of the priority queue and `:comparator` be an object implementing the `comparator` protocol. `:comparator` must be ordered.

returns a new `binary-heap` with a capacity of `:capacity`

plus 1 fill `binary-heap` `=/:capacity :comparator . items=/` [procedure]

requires `:capacity` be a positive integer specifying the maximum capacity of the priority queue, `:comparator` be an object implementing the `comparator` protocol, and `items` be a list of items to initialize the queue with. `:comparator` must be ordered.

returns a new `binary-heap` with a capacity of `:capacity` containing `items` enqueued on the priority queue. If the number of `items` is greater than `:capacity`, `&invalid-argument-exception` is thrown.

plus 1 fill `=binary-heap-copy` `=/=pqueue=/` [procedure]

requires `pqueue` be a `binary-heap`.

returns a shallow copy of `pqueue`.

plus 1 fill `=binary-heap-empty?` `=/=pqueue=/` [procedure]

requires `pqueue` be a `binary-heap`.

returns a boolean indicating whether or not the priority queue is empty.

plus 1 fill `=binary-heap-length` `=/=pqueue=/` [procedure]

requires `pqueue` be a `binary-heap`.

returns the length the priority queue.

plus 1 fill `=binary-heap-capacity /=pqueue=/` [procedure]

requires `pqueue` be a `binary-heap`.

returns the maximum size or length of `queue`.

plus 1 fill `=binary-heap-enqueue! /=pqueue item=/` [procedure]

requires `pqueue` be a `binary-heap` and `item` be an arbitrary object.

modifies `pqueue` by placing `item` on the priority queue.

returns `#unspecified` or if there is no free capacity, throws `&invalid-state-exception`.

plus 1 fill `=binary-heap-dequeue! /=pqueue=/` [procedure]

requires `pqueue` be a `binary-heap`.

modifies `pqueue` by removing the highest (or lowest, depending on the comparator) priority item from the priority queue.

returns the item removed from `pqueue` or if `pqueue` is empty, throws `&invalid-state-exception`.

plus 1 fill `=binary-heap-first /=pqueue=/` [procedure]

requires `pqueue` be a `binary-heap`.

returns the the highest (or lowest, depending on the comparator) priority item from `pqueue` or if `pqueue` is empty, throws `&invalid-state-exception`.

3.14.3 Examples

2 procedures are provided for creating a **binary-heap**: The first creates an empty queue and the other populates the queue with the items passed to it:

```
== (enumerable-collect (make-binary-heap :capacity 4 :comparator number-comparator)  
list-collector) => ()  
(enumerable-collect (binary-heap :capacity 4 :comparator number-comparator  
1 2 3) list-collector) => (1 2 3)
```

An item can be placed onto the priority queue with **binary-heap-enqueue!**:

```
== (let ((pqueue (binary-heap :capacity 4 :comparator number-comparator  
1 2 3))) (binary-heap-enqueue! pqueue 4) (binary-heap-first pqueue)) =>  
1
```

The highest (or lowest, depending on the comparator) priority item of the priority queue can be non-destructively obtained with **binary-heap-first**:

```
== (let ((pqueue (binary-heap :capacity 4 :comparator number-comparator  
1 2 3))) (binary-heap-first pqueue)) => 1
```

The highest (or lowest, depending on the comparator) priority item can be removed from the priority queue with **binary-heap-dequeue!**:

```
== (let ((pqueue (binary-heap :capacity 4 :comparator number-comparator  
1 2 3))) (binary-heap-dequeue! pqueue) (binary-heap-first pqueue)) =>  
2
```

To test if a priority queue is empty, use **binary-heap-empty?**:

```
== (binary-heap-empty? (binary-heap :capacity 4 :comparator number-comparator)  
=> #t  
(binary-heap-empty? (binary-heap :capacity 4 :comparator number-comparator  
1 2)) => #f
```

The size or length of a priority queue is obtained with **binary-heap-length**:

```
== (binary-heap-length (binary-heap :capacity 4 :comparator number-comparator  
1 2)) => 2
```

The capacity or maximum length of a priority queue is obtained with **binary-heap-capacity**:

```
== (binary-heap-capacity (binary-heap-capacity :capacity 4 :compara-  
tor number-comparator 1 2)) => 4
```

To make a shallow copy of a priority queue, use **binary-heap-copy**:

```
== (let* ((pqueue1 (binary-heap :capacity 4 :comparator number-comparator  
1 2)) (pqueue2 (binary-heap-copy queue1))) (eq? queue1 queue2)) =>  
#f
```

`binary-heap` also implements the `priority-queue`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section [\[\[#anchor:priority002dqueue-Examples\]\[2.14.3 \[priority-queue Examples\], page \]\]](#). See Section [\[\[#anchor:collection-Examples\]\[2.1.3 \[collection Examples\], page \]\]](#). See Section [\[\[#anchor:extendable-Examples\]\[2.3.3 \[extendable Examples\], page \]\]](#). See Section [\[\[#anchor:mutable-Examples\]\[2.4.3 \[mutable Examples\], page \]\]](#). See Section [\[\[#anchor:enumerable-Examples\]\[2.6.3 \[enumerable Examples\], page \]\]](#).

3.15 pairing-heap

3.15.1 Overview

3.15.2 API Reference

`pairing-heap` implements the `priority-queue`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section [\[\[#anchor:priority002dqueue\]\[2.14 \[priority-queue\], page \]\]](#). See Section [\[\[#anchor:collection\]\[2.1 \[collection\], page \]\]](#). See Section [\[\[#anchor:extendable\]\[2.3 \[extendable\], page \]\]](#). See Section [\[\[#anchor:mutable\]\[2.4 \[mutable\], page \]\]](#). See Section [\[\[#anchor:enumerable\]\[2.6 \[enumerable\], page \]\]](#).

`plus 1 fill =pairing-heap? =/object=/` [procedure]

returns a boolean indicating whether or not `object` is a pairing-heap.

`plus 1 fill make-pairing-heap =/:comparator=/` [procedure]

requires `:comparator` be an object implementing the `comparator` protocol. `:comparator` must be ordered.

returns a new `pairing-heap`.

`plus 1 fill pairing-heap =/:comparator . items=/` [procedure]

requires `:comparator` be an object implementing the `comparator` protocol and `items` be a list of items to initialize the priority queue with. `:comparator` must be ordered.

returns a new `pairing-heap` containing `items` enqueued from left-to-right onto the priority queue.

plus 1 fill =pairing-heap-copy /=queue=/ [procedure]

requires queue be a **pairing-heap**.

returns returns a shallow copy of queue.

plus 1 fill =pairing-heap-empty? /=pqueue=/ [procedure]

requires pqueue be a **pairing-heap**.

returns returns a boolean indicating whether or not the priority queue is empty.

plus 1 fill =pairing-heap-length /=pqueue=/ [procedure]

requires pqueue be a **pairing-heap**.

returns returns the length the priority queue.

plus 1 fill =pairing-heap-enqueue! /=pqueue item=/ [procedure]

requires pqueue be a **pairing-heap** and item be an arbitrary object.

modifies pqueue by placing item on the priority queue.

returns #unspecified.

plus 1 fill =pairing-heap-dequeue! /=pqueue=/ [procedure]

requires pqueue be a **pairing-heap**.

modifies pqueue by removing the highest(or lowest, depending on the comparator) priority item from the priority queue.

returns the item removed from pqueue or if pqueue is empty, throws &invalid-state-exception.

plus 1 fill =pairing-heap-first /=pqueue=/ [procedure]

requires `pqueue` be a `pairing-heap`.

returns the highest(or lowest, depending on the comparator) item from `pqueue` or if `pqueue` is empty, throws `&invalid-state-exception`.

3.15.3 Examples

2 procedures are provided for creating a `pairing-heap`: The first creates an empty queue and the other populates the queue with the items passed to it:

```
== (enumerable-collect (make-pairing-heap :comparator number-comparator)  
list-collector) => ()  
(enumerable-collect (pairing-heap :comparator number-comparator 1 2  
3) list-collector) => (1 3 2)
```

An item can be placed onto the priority queue with `pairing-heap-enqueue!`:

```
== (let ((pqueue (pairing-heap :comparator number-comparator 1 2 3)))  
(pairing-heap-enqueue! pqueue 4) (pairing-heap-first pqueue)) => 1
```

The highest(or lowest, depending on the comparator) priority item of the queue can be non-destructively obtained with `pairing-heap-first`:

```
== (let ((pqueue (pairing-heap :comparator number-comparator 1 2 3)))  
(pairing-heap-first pqueue)) => 1
```

The highest(or lowest, depending on the comparator) priority item can be removed from the priority queue with `pairing-heap-dequeue!`:

```
== (let ((pqueue (pairing-heap :comparator number-comparator 1 2 3)))  
(pairing-heap-dequeue! pqueue) (pairing-heap-first pqueue)) => 2
```

To test if a priority queue is empty, use `pairing-heap-empty?`:

```
== (pairing-heap-empty? (pairing-heap :comparator number-comparator)  
=> #t  
(pairing-heap-empty? (pairing-heap :comparator number-comparator 1  
2)) => #f
```

The size or length of a priority queue is obtained with `pairing-heap-length`:

```
== (pairing-heap-length (pairing-heap :comparator number-comparator  
1 2)) => 2
```

To make a shallow copy of a priority queue, use `pairing-heap-copy`:

```
== (let* ((pqueue1 (pairing-heap :comparator number-comparator 1 2))  
(pqueue2 (pairing-heap-copy queue1))) (eq? pqueue1 pqueue2)) => #f
```

`pairing-heap` also implements the `priority-queue`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section [\[\[#anchor:priority002dqueue-](#)

Examples][2.14.3 [priority-queue Examples], page]]. See Section [[#anchor:collection-Examples][2.1.3 [collection Examples], page]]. See Section [[#anchor:extendable-Examples][2.3.3 [extendable Examples], page]]. See Section [[#anchor:mutable-Examples][2.4.3 [mutable Examples], page]]. See Section [[#anchor:enumerable-Examples][2.6.3 [enumerable Examples], page]].

3.16 sorted-dictionary

3.16.1 Overview

`sorted-dictionary` is a balanced-tree based implementation of the `dictionary` protocol.

3.16.2 API Reference

`sorted-dictionary` implements the `dictionary`, `collection`, `extendable`, `indexable`, `mutable`, and `enumerable` protocols. See Section [[#anchor:dictionary][2.15 [dictionary], page]]. See Section [[#anchor:collection][2.1 [collection], page]]. See Section [[#anchor:extendable][2.3 [extendable], page]]. See Section [[#anchor:indexable][2.2 [indexable], page]]. See Section [[#anchor:mutable][2.4 [mutable], page]]. See Section [[#anchor:enumerable][2.6 [enumerable], page]].

plus 1 fill `=sorted-dictionary? =/object=/` [procedure]

returns a boolean indicating whether or not `object` is a sorted-dictionary.

plus 1 fill `make-sorted-dictionary =/:comparator=/` [procedure]

requires `:comparator` be an object supporting the `comparator` protocol. `:comparator` must be ordered.

returns a sorted-dictionary.

plus 1 fill `sorted-dictionary =/:comparator . list-of-associations=/` [procedure]

requires `:comparator` be an object supporting the `comparator` protocol and `list-of-associations` be a list of `association=s`. `=:comparator` must be ordered.

returns a sorted-dictionary.

plus 1 fill =sorted-dictionary-get /=dict key=/ [procedure]

requires `dict` be a `sorted-dictionary` and `key` be an arbitrary object. `key` must be supported by the `comparator` used when creating `dict`

returns the value associated with `key` or `#f` if no such association exists.

plus 1 fill =sorted-dictionary-put! /=dict key value=/ [procedure]

requires `dict` be a `sorted-dictionary` `key` and `value` be arbitrary objects. `key` must be supported by the `comparator` used when creating `dict`.

modifies `dict` so that it contains the association of `key` to `value`. If an association with `key` already exists it is replaced, and if not, a new association is created.

returns `#unspecified`

plus 1 fill =sorted-dictionary-update! /=dict key value exist-fun=/ [procedure]

requires `dict` be a `sorted-dictionary`, `key` and `value` be arbitrary objects, and `exist-fun` which is a procedure excepting a single value and returning an updated value when an association with a key `key` already exists. `key` must be supported by the `comparator` used when creating `dict`.

modifies `dict` such that if the `sorted-dictionary` doesn't currently contain an association with a key `key`, it contains an association with key `key` and value `value`, or if an existing association exists updates it so that has the value obtained by applying `exist-fun` to its value.

returns `#unspecified`

plus 1 fill =sorted-dictionary-contains? /=dict key=/ [procedure]

requires `dict` be a `sorted-dictionary` and `key` be an arbitrary object.

returns a boolean indicating whether or not `dict` contains an association with the key `key`.

plus 1 fill `=sorted-dictionary-remove! =/=dict key=/` [procedure]

requires `dict` be a `sorted-dictionary` and `key` be an arbitrary object. `key` must be supported by the `comparator` used when creating `dict`

modifies `dict` by removing the association with the key `key`.

returns `#unspecified`

plus 1 fill `=sorted-dictionary-empty? =/=dict=/` [procedure]

requires `dict` be `sorted-dictionary`.

returns a boolean indicating whether or not the `sorted-dictionary` contains any associations.

plus 1 fill `=sorted-dictionary-copy =/=dict=/` [procedure]

requires `dict` be a `sorted-dictionary`.

returns a shallow copy of `dict`

plus 1 fill `=sorted-dictionary-length =/=dict=/` [procedure]

requires `dict` be a `sorted-dictionary`.

returns the number of items in `dict`.

3.16.3 Examples

To test whether an object is a `sorted-dictionary` use the predicate `sorted-dictionary?`:

```
== (sorted-dictionary? (sorted-dictionary :comparator string-comparator))  
⇒ #t  
(sorted-dictionary? (vector)) ⇒ #f
```

Two procedures are used to create `sorted-dictionary=s`. The first creates an empty dictionary, and the second allows for the creation of a dictionary with the provided values. Both require a `=:comparator` argument supporting the type of the key. The `:comparator` must be ordered.

```
== (let ((dict (make-sorted-dictionary :comparator string-comparator)))  
    (sorted-dictionary? dict)) ⇒ #t  
(let ((dict (sorted-dictionary :comparator string-comparator (=> "a" 1)  
                                (=> "b" 2)))) (sorted-dictionary-length dict)) ⇒ 2
```

Checking whether a dictionary is empty is accomplished with `sorted-dictionary-empty?`:

```
== (sorted-dictionary-empty? (sorted-dictionary :comparator string-comparator))  
⇒ #t  
(sorted-dictionary-empty? (sorted-dictionary :comparator string-comparator  
                                (=> "A" 1))) ⇒ #f
```

Associations can be added to a dictionary with `sorted-dictionary-put!`:

```
== (let ((dict (sorted-dictionary :comparator string-comparator))) (sorted-  
dictionary-put! dict "a" 1) (sorted-dictionary-put! dict "b" 2) (map (lambda  
(kv) (cons (=>key kv) (=>value kv))) (dictionary-enumerable-collect dict  
list-collector))) ⇒ (("a" . 1) ("b" . 2))
```

And removed (assuming the insertions above) with `sorted-dictionary-remove!`:

```
== (sorted-dictionary-remove! dict "a") (map (lambda (kv) (cons (=>key  
kv) (=>value kv))) (dictionary-enumerable-collect dict list-collector)) ⇒  
(("b" . 2))
```

The value associated with a given key is easily obtained:

```
== (let ((dict (sorted-dictionary :comparator string-comparator (=>  
"a" 1) (=> "b" 2) (=> "c" 3)))) (sorted-dictionary-get dict "b") ⇒ 2
```

The `dictionary-update!` method can be used to update an existing value or insert a new value if an existing association does not exist:

```
== (let ((dict (sorted-dictionary :comparator string-comparator (=>  
"a" 1) (=> "b" 2) (=> "c" 3)))) (sorted-dictionary-update! dict "b" 0
```

```
(lambda (x) (+ x 1))) (sorted-dictionary-update! dict "d" 0 (lambda (x)
(+ x 1))) (sorted-dictionary-get dict "b") => 3 (sorted-dictionary-get
dict "d") => 0
```

To obtain the number of associations in a dictionary, call **sorted-dictionary-length**:

```
== (let ((dict (sorted-dictionary :comparator string-comparator (= >
"a" 1) (= > "b" 2) (= > "c" 3)))) (sorted-dictionary-length dict) => 3
```

Querying whether an association with a given key is accomplished with **sorted-dictionary-contains?**:

```
== (let ((dict (sorted-dictionary :comparator string-comparator (= >
"a" 1) (= > "b" 2) (= > "c" 3)))) (sorted-dictionary-contains? dict "a")
=> #t
(sorted-dictionary-contains? dict "d") => #f
```

Copying a dictionary is accomplished with **dictionary-copy**:

```
== (let* ((dict1 (sorted-dictionary :comparator string-comparator (= >
"a" 1) (= > "b" 2) (= > "c" 3))) (dict2 (sorted-dictionary-copy dict1)))
(eq? dict1 dict2) => #f (and (sorted-dictionary-contains? dict1 "a")
(sorted-dictionary-contains? dict2 "a")) => #t
```

sorted-dictionary also implements the **dictionary**, **collection**, **mutable**, **indexable**, **extendable**, **enumerable**, and **dictionary-enumerable** protocols. See Section [\[\[#anchor:dictionary\]\[2.15 \[dictionary\], page \]\]](#). See Section [\[\[#anchor:collection-Examples\]\[2.1.3 \[collection Examples\], page \]\]](#). See Section [\[\[#anchor:mutable-Examples\]\[2.4.3 \[mutable Examples\], page \]\]](#). See Section [\[\[#anchor:indexable-Examples\]\[2.2.3 \[indexable Examples\], page \]\]](#). See Section [\[\[#anchor:extendable-Examples\]\[2.3.3 \[extendable Examples\], page \]\]](#). See Section [\[\[#anchor:enumerable-Examples\]\[2.6.3 \[enumerable Examples\], page \]\]](#). See Section [\[\[#anchor:dictionary-enumerable-Examples\]\[2.17.3 \[dictionary-enumerable Examples\], page \]\]](#).

4 Global Index