Package 'Immutables'

May 7, 2026

Type Package

Title Fast and Functional Data Structures

URL https://oneilsh.github.io/immutables/,
https://github.com/oneilsh/immutables

BugReports https://github.com/oneilsh/immutables/issues

Version 1.0.1

Description Provides fast, side-effect free data structures, including catenable named lists, priority queues, double-ended queues, ordered sequences, and interval indices. Implementation is based on the finger-tree data structure of Hinze and Paterson (2006) <doi:10.1017/S0956796805005769>.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 4.1.0)

Imports coro, Rcpp, lambda.r

LinkingTo Rcpp

Suggests covr, ggplot2, ggtext, igraph, IRanges, knitr, microbenchmark, pkgdown, pkgload, rmarkdown, rprojroot, rstackdeque, rticles, S4Vectors, scales, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

RoxygenNote 7.3.3

NeedsCompilation yes

Author Shawn T. O'Neil [aut, cre]

Maintainer Shawn T. O'Neil <shawn@tislab.org>

Repository CRAN

Date/Publication 2026-04-28 20:10:13 UTC

Contents

add_monoids . . . 4
as.list.flexseq . . . 5
as.list.interval_index . . . 6
as.list.ordered_sequence . . . 6
as.list.priority_queue . . . 7
as_flexseq . . . 8
as_interval_index . . . 9
as_iterator.flexseq . . . 10
as_iterator.priority_queue . . . 11
as_ordered_sequence . . . 12
as_priority_queue . . . 13
c.flexseq . . . 13
count_between . . . 14
count_key . . . 15
elements_between . . . 16
fapply . . . 17
flexseq . . . 18
get_measure . . . 19
get_measures . . . 20
insert . . . 21
insert_at . . . 22
interval_index . . . 23
length.flexseq . . . 24
length.interval_index . . . 24
length.ordered_sequence . . . 25
length.priority_queue . . . 26
locate_by_predicate . . . 26
loop . . . 28
lower_bound . . . 28
max_endpoint . . . 29
max_key . . . 30
max_priority . . . 30
measure_monoid . . . 31
merge.flexseq . . . 32
merge.interval_index . . . 33
merge.ordered_sequence . . . 34
merge.priority_queue . . . 35
min_endpoint . . . 36
min_key . . . 37
min_priority . . . 37
ordered_sequence . . . 38
peek_all_containing . . . 39
peek_all_key . . . 39
peek_all_max . . . 40
peek_all_min . . . 41
peek_all_overlaps . . . 41

peek_all_point . . . 42
peek_all_within . . . 43
peek_at . . . 44
peek_back . . . 44
peek_containing . . . 45
peek_front . . . 46
peek_key . . . 46
peek_max . . . 47
peek_min . . . 48
peek_overlaps . . . 48
peek_point . . . 49
peek_within . . . 50
plot_structure . . . 51
pop_all_containing . . . 53
pop_all_key . . . 54
pop_all_max . . . 55
pop_all_min . . . 55
pop_all_overlaps . . . 56
pop_all_point . . . 57
pop_all_within . . . 58
pop_at . . . 59
pop_back . . . 60
pop_containing . . . 61
pop_front . . . 61
pop_key . . . 62
pop_max . . . 63
pop_min . . . 64
pop_overlaps . . . 65
pop_point . . . 65
pop_within . . . 66
print.flexseq . . . 67
print.interval_index . . . 68
print.ordered_sequence . . . 69
print.priority_queue . . . 70
priority_queue . . . 70
push_back . . . 71
push_front . . . 72
split_around_by_predicate . . . 73
split_at . . . 74
split_by_predicate . . . 75
sub-.interval_index . . . 76
sub-.ordered_sequence . . . 77
sub-.priority_queue . . . 78
unlist.flexseq . . . 80
upper_bound . . . 81
validate_name_state . . . 82
validate_tree . . . 82
$.flexseq . . . 83

Index **86**

`add_monoids` *Add or Merge Measure Monoids*

Description

Attaches one or more named `measure_monoid()` definitions to an existing immutable structure.

Usage

```
add_monoids(t, monoids, overwrite = FALSE)
```

Arguments

`t`	Immutable structure (`flexseq` and subclasses).
`monoids`	Named list of `measure_monoid()` objects.
`overwrite`	Logical; if `TRUE`, replace existing monoids with the same names. If `FALSE`, existing names are kept.

Details

Mechanics:

- Each monoid name defines an independent accumulated measure over elements in the tree.
- New monoids are computed for all elements and cached in the returned object.
- Existing monoids are unchanged unless `overwrite = TRUE`.
- Structural/reserved monoid names cannot be replaced.

Measure-function signatures:

- `flexseq`: `measure(entry)` where `entry` is the stored element.
- `ordered_sequence`: `measure(entry)` where `entry` is `list(value, key)`.
- `priority_queue`: `measure(entry)` where `entry` is `list(value, priority)`.
- `interval_index`: `measure(entry)` where `entry` is `list(value, start, end)`.

This operation is persistent: `t` is not modified.

Use this when you want fast predicate scans (for example with `locate_by_predicate()`, `split_by_predicate()`, `split_around_by_predicate()`) driven by domain-specific accumulated values.

Value

A persistent copy with updated monoid definitions and cached measures.

See Also

`measure_monoid()`, `get_measure()`, `get_measures()`

Examples

```
x <- flexseq(10, 20, 30)

running_sum <- measure_monoid(`+`, 0, as.numeric)
x2 <- add_monoids(x, list(sum = running_sum))
attr(x2, "measures")$sum

# Use the monoid in a split query
split_around_by_predicate(x2, function(v) v >= 30, "sum")

# Overwrite an existing monoid definition
running_count <- measure_monoid(`+`, 0L, function(e) 1L)
x3 <- add_monoids(x2, list(sum = running_count), overwrite = TRUE)
attr(x3, "measures")$sum
```

---

as.list.flexseq *Coerce a Sequence to Base List*

Description

Returns elements in left-to-right sequence order.

Usage

```
## S3 method for class 'flexseq'
as.list(x, ...)
```

Arguments

x	A flexseq.
...	Unused.

Details

Returns payload elements in sequence order. If the sequence is fully named, those names are preserved on the returned list.

Value

A base R list of sequence elements.

Examples

```
x <- flexseq("a", "b", "c")
as.list(x)

n <- flexseq(a = 1, b = 2)
as.list(n)
```

as.list.interval_index

Coerce Interval Index to List

Description

Coerce Interval Index to List

Usage

```
## S3 method for class 'interval_index'
as.list(x, ...)
```

Arguments

x	An interval_index.
...	Unused.

Details

This returns payload values only.

Value

A plain list of payload elements in interval order.

Examples

```
ix <- interval_index("a", "b", "c", start = c(3, 1, 2), end = c(4, 2, 3))
as.list(ix)
```

as.list.ordered_sequence

Coerce Ordered Sequence to List

Description

Coerce Ordered Sequence to List

Usage

```
## S3 method for class 'ordered_sequence'
as.list(x, ...)
```

Arguments

x	An ordered_sequence.
...	Unused.

Details

Returns payload elements only (keys are omitted) in canonical key order. If entries are named, names are preserved on the returned list.

Value

A plain list of elements in key order.

Examples

```
x <- ordered_sequence("a", "b", "c", keys = c(2, 1, 3))
as.list(x)
```

---

as.list.priority_queue

Coerce Priority Queue to List

Description

Returns queue entries as a plain list of records with fields value and priority, in queue sequence order.

Usage

```
## S3 method for class 'priority_queue'
as.list(x, ..., drop_meta = FALSE)
```

Arguments

x	A priority_queue.
...	Unused.
drop_meta	Logical scalar. When FALSE (default), returns full queue entry records (value + priority). When TRUE, returns payload values only.

Details

Each returned entry is a record with fields value and priority. Entry names (when present) are preserved on the returned list.

Value

A plain list of queue entry records (drop_meta = FALSE) or payload values (drop_meta = TRUE).

Examples

```
q <- priority_queue("a", "b", priorities = c(2, 1))
as.list(q)
as.list(q, drop_meta = TRUE)
```

---

as_flexseq *Coerce Objects to* flexseq

Description

as_flexseq() is the canonical way to obtain a plain flexseq for full sequence-style operations.

Usage

```
as_flexseq(x)
```

Arguments

x	Input object.

Details

For base vectors/lists, this builds a new flexseq preserving element order and names.

For specialized immutable subclasses (priority_queue, ordered_sequence, interval_index), this intentionally drops subclass semantics and returns a plain flexseq.

This is an S3 generic. Notable method behavior:

- as_flexseq.flexseq(x) returns x unchanged.
- as_flexseq.priority_queue(x) returns payload items.
- as_flexseq.ordered_sequence(x) returns payload items.
- as_flexseq.interval_index(x) returns payload items.

For advanced types, custom monoids are dropped and the rebuilt flexseq keeps only structural monoids (.size, .named_count). For priority_queue, a flexseq of full entry records can be obtained by composing with as.list(): as_flexseq(as.list(x)). For ordered_sequence and interval_index, as.list() also returns payload-only lists, so no direct record-preserving cast is provided.

Value

A plain flexseq.

See Also

flexseq(), priority_queue(), ordered_sequence(), interval_index()

Examples

```
x <- as_flexseq(1:3)
x

q <- priority_queue("a", "b", priorities = c(2, 1))
as_flexseq(q)

o <- ordered_sequence("a", "b", keys = c(2, 1))
as_flexseq(o)
```

---

as_interval_index *Build an Interval Index from* x*,* start*, and* end

---

Description

Constructs an interval_index by pairing each element of x with corresponding start and end endpoints.

Usage

```
as_interval_index(x, start, end, default_query_bounds = "[)")
```

Arguments

x	Elements to add.
start	Start endpoints with the same length as x.
end	End endpoints with the same length as x.
default_query_bounds	Boundary convention used as the default for query operations on this index: one of "[)", "[]", "()", "(]". Per-query peek_* / pop_* calls may override via their own bounds argument.

Details

Output is ordered by interval start.

Names on x are preserved as element names.

Value

An interval_index.

Examples

```
ix <- as_interval_index(c("a", "b", "c"), start = c(1, 2, 2), end = c(3, 2, 4))
ix
as.list(peek_all_point(ix, 2))

# Endpoints can be other comparable types
ix_date <- as_interval_index(
  c("phase1", "phase2"),
  start = as.Date(c("2024-01-01", "2024-01-10")),
  end = as.Date(c("2024-01-05", "2024-01-15"))
)
ix_date
```

`as_iterator.flexseq` *Iterate over a* `flexseq` *(coro iterator)*

Description

Returns a lazy iterator that yields payload elements left-to-right. Use with `loop()` as the canonical iteration form:

Usage

```
## S3 method for class 'flexseq'
as_iterator(x)
```

Arguments

`x`	A `flexseq`.

Details

```
loop(for (x in s) print(x))
```

Iteration uses repeated left-view (`viewL`) and is O(n) total, O(1) amortized per step. The original `x` is not modified; the iterator holds a private cursor over progressively-smaller tails.

For named sequences, internal name metadata is stripped from yielded values to match `peek_front(s)` semantics. Access names via `as.list()` when needed.

Inherited by `ordered_sequence` and `interval_index`: for those subclasses the yielded value is the unwrapped payload (keys / interval endpoints dropped), in key-ascending / start-position order respectively. See `as_iterator.priority_queue()` for the priority-order override.

Value

A `coro` iterator function.

Do not use plain `for` directly

Writing `for (x in s) ...` (without `loop()`) will not dispatch this method. R's `for` walks the object's underlying list storage at the C level and bypasses S3 `length`/`[[`, so it silently yields raw finger-tree internals (Digit/Empty/Deep nodes) rather than sequence elements. Always wrap with `loop()`, or call `as.list()` first for an eager copy.

Examples

```
s <- flexseq("a", "b", "c")
loop(for (x in s) print(x))
```

---

as_iterator.priority_queue

Iterate over a `priority_queue` *(coro iterator)*

Description

Returns a lazy iterator that yields payload elements in priority-ascending order. Use with `loop()`:

Usage

```
## S3 method for class 'priority_queue'
as_iterator(x)
```

Arguments

`x`	A `priority_queue`.

Details

```
loop(for (x in pq) print(x))
```

Traversal is driven by repeated `pop_min()`: each step is O(log n), so full traversal is O(n log n). Ties within equal priorities are yielded in FIFO insertion order (inherited from `pop_min()`).

The original `x` is not modified; the iterator holds a private cursor and partial iteration (e.g. via `break`) leaves the source intact.

Each yielded value is the bare payload (matching `peek_min()`). Use `fapply()` if your callback needs the priority alongside the value, or cast with `as_flexseq()` for insertion-order iteration.

Value

A `coro` iterator function.

Examples

```
pq <- priority_queue("a", "b", "c", priorities = c(3, 1, 2))
loop(for (x in pq) print(x))  # "b", "c", "a"
```

as_ordered_sequence *Build an Ordered Sequence from* x *and* keys

Description

Constructs an ordered_sequence by pairing each element of x with the corresponding key in keys.

Usage

```
as_ordered_sequence(x, keys)
```

Arguments

x	Elements to add.
keys	Key values with the same length as x.

Details

Output is always sorted by key.

Duplicate keys are allowed; ties preserve input order (stable/FIFO within the same key).

Names on x are preserved as element names.

Value

An ordered_sequence.

Examples

```
xs <- as_ordered_sequence(c("d", "a", "b", "a2"), keys = c(4, 1, 2, 1))
xs
length(elements_between(xs, 1, 1))

n <- as_ordered_sequence(setNames(as.list(c("a", "b")), c("ka", "kb")), keys = c(2, 1))
n[["kb"]]

# Keys can be other comparable types
num_by_chr <- as_ordered_sequence(c(20, 10, 30), keys = c("b", "a", "c"))
num_by_chr
```

as_priority_queue *Build a Priority Queue from* x *and* priorities

Description

Constructs a queue by pairing each element of x with the corresponding value in priorities.

Usage

```
as_priority_queue(x, priorities)
```

Arguments

x	Elements to enqueue.
priorities	Priorities with the same length as x.

Details

x is interpreted element-wise (via list coercion). Names on x are preserved as queue element names.

All priorities must be non-missing and mutually comparable.

Value

A priority_queue.

Examples

```
x <- as_priority_queue(letters[1:4], priorities = c(3, 1, 2, 1))
x
peek_min(x)
peek_max(x)

# Names are preserved
n <- as_priority_queue(setNames(as.list(1:3), c("a", "b", "c")), priorities = c(2, 1, 3))
n[["b"]]
```

c.flexseq *Concatenate Sequences*

Description

Concatenate Sequences

Usage

```
## S3 method for class 'flexseq'
c(..., recursive = FALSE)
```

Arguments

...	flexseq objects.
recursive	Unused; must be FALSE.

Details

c() is supported for flexseq and returns a new concatenated flexseq.

For priority_queue, ordered_sequence, and interval_index, c() is not supported because concatenation can violate structure-specific invariants. Cast first with as_flexseq() when sequence-style concatenation is intended, noting that this drops ordering and priority metadata.

Value

A concatenated flexseq.

Examples

```
x <- flexseq("a", "b")
y <- flexseq("c", "d")
c(x, y)

q1 <- priority_queue("a", priorities = 2)
q2 <- priority_queue("b", priorities = 1)
try(c(q1, q2))
c(as_flexseq(q1), as_flexseq(q2))

o1 <- ordered_sequence("a", keys = 1)
o2 <- ordered_sequence("b", keys = 2)
try(c(o1, o2))
```

---

count_between *Count Elements in a Key Range*

Description

Count Elements in a Key Range

Usage

```
count_between(x, from_key, to_key, include_from = TRUE, include_to = TRUE)
```

Arguments

x	An ordered_sequence.
from_key	Lower bound key.
to_key	Upper bound key.
include_from	Include lower bound when TRUE.
include_to	Include upper bound when TRUE.

Details

Uses the same range semantics as `elements_between()` but returns only the count:

- `include_from = TRUE` uses `key >= from_key`; otherwise `key > from_key`.
- `include_to = TRUE` uses `key <= to_key`; otherwise `key < to_key`.

Value

Integer count of matches.

Examples

```
x <- ordered_sequence("a", "b", "c", "d", keys = c(1, 2, 2, 3))
count_between(x, 2, 3)
count_between(x, 2, 2, include_to = FALSE)
```

`count_key` *Count Elements Matching One Key*

Description

Count Elements Matching One Key

Usage

```
count_key(x, key)
```

Arguments

`x`	An `ordered_sequence`.
`key`	Query key.

Details

Counts multiplicity for a single key. Returns `0L` when the key is not present.

Value

Integer count of matches.

Examples

```
x <- ordered_sequence("a", "b", "c", keys = c(1, 2, 2))
count_key(x, 2)
count_key(x, 10)
```

elements_between *Return Elements in a Key Range*

Description

Return Elements in a Key Range

Usage

```
elements_between(x, from_key, to_key, include_from = TRUE, include_to = TRUE)
```

Arguments

x	An ordered_sequence.
from_key	Lower bound key.
to_key	Upper bound key.
include_from	Include lower bound when TRUE.
include_to	Include upper bound when TRUE.

Details

Range membership is controlled by include_from and include_to:

- include_from = TRUE uses key >= from_key; otherwise key > from_key.
- include_to = TRUE uses key <= to_key; otherwise key < to_key.

If no elements fall in the range, returns an empty ordered_sequence.

Value

An ordered_sequence of matched elements, in key order. Use as.list() to convert to a plain list.

Examples

```
x <- ordered_sequence("a", "b", "c", "d", keys = c(1, 2, 2, 3))
elements_between(x, 2, 3)
as.list(elements_between(x, 2, 3))
elements_between(x, 2, 2, include_to = FALSE)
```

`fapply` *Fapply with S3 dispatch*

Description

`fapply()` is an S3 generic for applying functions over immutable structures with type-specific dispatch.

Usage

```
fapply(X, FUN, ...)
```

Arguments

`X`	Object to apply over.
`FUN`	Function to apply.
`...`	Method-specific arguments.

Details

`fapply()` returns a new object and preserves class semantics.

Method signatures:

- `fapply.flexseq(X, FUN, ..., preserve_custom_monoids = TRUE)`
- `fapply.priority_queue(X, FUN, ..., preserve_custom_monoids = TRUE)`
- `fapply.ordered_sequence(X, FUN, ..., preserve_custom_monoids = TRUE)`
- `fapply.interval_index(X, FUN, ..., preserve_custom_monoids = TRUE)`

For `priority_queue`, `ordered_sequence`, and `interval_index`, `FUN` receives structured fields (`value` plus metadata) and should return the new payload value only; ordering metadata is preserved. The name argument is optional: callbacks that only take value/metadata also work.

If supported by the method, `preserve_custom_monoids = TRUE` keeps added user monoids; `FALSE` rebuilds with required structural monoids only.

Value

Method-dependent result.

See Also

`flexseq()`, `priority_queue()`, `ordered_sequence()`, `interval_index()`

Examples

```
x <- flexseq("a", "b", "c")
fapply(x, toupper)

q <- priority_queue(one = "a", two = "b", priorities = c(2, 1))
fapply(q, function(value, priority, name) paste0(value, priority))

o <- ordered_sequence(one = "a", two = "b", keys = c(2, 1))
fapply(o, function(value, key, name) paste0(value, "_", key))

ix <- interval_index(one = "a", two = "b", start = c(1, 3), end = c(2, 4))
fapply(ix, function(value, start, end, name) paste0(value, "[", start, ",", end, "]"))
```

`flexseq` *Construct a Persistent Flexible Sequence*

Description

`flexseq(...)` creates an immutable sequence from `...`, preserving element order and optional names, with efficient persistent updates.

Usage

```
flexseq(...)
```

Arguments

`...`	Sequence elements.

Details

It is list-like in payload flexibility (any R object per element), but sequence-oriented in API (`push_*`, `peek_*`, `pop_*`, indexing, split/concat).

`flexseq` is the base general-purpose structure in this package. Specialized structures such as `priority_queue`, `ordered_sequence`, and `interval_index` build on related internals but expose narrower semantics.

`flexseq` operations are persistent: updates return new objects and do not mutate prior versions.

Value

A `flexseq` object.

See Also

`as_flexseq()`, `priority_queue()`, `ordered_sequence()`, `interval_index()`

Examples

```
x <- flexseq(1, 2, 3)
x

y <- push_front(x, 0)
y
x  # unchanged

named <- flexseq(a = 1, b = 2)
named
named[["a"]]
```

`get_measure` *Read a Cached Subtree Measure*

Description

Returns the monoid's accumulated measure across the whole (sub)tree at the root of `x`. This is the cached value that internal operations use for O(log n) locate and split.

Usage

```
get_measure(x, monoid_name)
```

Arguments

`x`	A flexseq (or any immutables subclass).
`monoid_name`	Name of an attached monoid (e.g. `.size`, a custom name from `add_monoids()`, or a built-in like `.pq_min` / `.oms_max_key`).

Details

On the full tree `x`, this is the aggregate over every element. After a split — e.g. `s <- split_by_predicate(x, p, "sum")` — call `get_measure(s$left, "sum")` to read the aggregate of just the left side in O(1).

For per-element measures, see `get_measures()`.

Value

The cached monoid value at the root of `x`. Shape depends on the monoid — typically atomic (e.g. a numeric sum) but may be a list for built-ins that carry auxiliary state.

See Also

`get_measures()`, `measure_monoid()`, `add_monoids()`

Examples

```
sum_m <- measure_monoid(`+`, 0, function(el) el)
x <- add_monoids(as_flexseq(c(3, 1, 4, 1, 5, 9, 2, 6)),
                 list(sum = sum_m))
get_measure(x, "sum")         # total across the whole tree
get_measure(x, ".size")       # built-in: element count

q <- priority_queue("a", "b", "c", priorities = c(5, 1, 3))
get_measure(q, ".pq_min")     # built-in, list-valued
```

---

get_measures *Read Per-Element Monoid Measures*

Description

Applies the named monoid's `measure()` function to each leaf entry in sequence order, returning the results as a new `flexseq`.

Usage

```
get_measures(x, monoid_name)
```

Arguments

`x`	A flexseq (or any immutables subclass).
`monoid_name`	Name of an attached monoid.

Details

Each leaf in a finger tree stores a structure-specific *entry* (flexseq: the raw element; ordered_sequence: `list(value, key)`; priority_queue: `list(value, priority)`; interval_index: `list(value, start, end)`). A monoid's `measure()` function is defined over that entry shape, and this accessor exposes the per-element values that would be combined to produce the cached aggregate.

Returns a `flexseq` rather than a base list so results can be piped back into other immutables operations; use `as.list()` or `unlist()` to convert.

Value

A plain unnamed `flexseq` of length `length(x)`. Entry `i` is the monoid's `measure()` applied to the `i`-th leaf entry.

See Also

`get_measure()`, `measure_monoid()`, `fapply()`

Examples

```
sum_m <- measure_monoid(`+`, 0, function(el) el)
x <- add_monoids(as_flexseq(c(3, 1, 4, 1, 5, 9)), list(sum = sum_m))
get_measures(x, "sum") |> unlist()

q <- priority_queue("a", "b", "c", priorities = c(5, 1, 3))
# Per-element .pq_min measures — each is list(has, priority).
get_measures(q, ".pq_min")
```

`insert` *Insert an Element*

Description

Inserts an element into a structure-specific position according to class semantics.

Usage

```
insert(x, ...)
```

Arguments

`x`	Object to insert into.
`...`	Method-specific arguments.

Details

`insert()` is an S3 generic. Required arguments in `...` depend on `x`:

- `priority_queue`: `element`, `priority` (optional `name`)
- `ordered_sequence`: `element`, `key` (optional `name`)
- `interval_index`: `element`, `start`, `end` (optional `name`)

This operation is persistent: `x` is not modified.

Value

Updated object of the same class as `x`.

See Also

`priority_queue()`, `ordered_sequence()`, `interval_index()`, `insert_at()`

Examples

```
q <- priority_queue("a", "b", priorities = c(2, 1))
insert(q, "c", priority = 3)

o <- ordered_sequence("a", "c", keys = c(1, 3))
insert(o, "b", key = 2)

iv <- interval_index("A", "B", start = c(1, 5), end = c(3, 8))
insert(iv, "C", start = 2, end = 6)
```

---

insert_at *Insert Elements at a Position*

Description

Inserts `values` before the current element at `index`.

Usage

```
insert_at(x, index, values)
```

Arguments

`x`	A `flexseq`.
`index`	One-based insertion position in `[1, length(x) + 1]`.
`values`	Values to insert.

Details

`values` is interpreted as a collection of elements to splice in.

Common cases:

- Atomic vector (`c("x", "y")`): inserts one element per vector entry.
- List (`list("x", "y")`): inserts one element per list entry.
- `flexseq`: inserts all of its elements.
- Empty input (`list()` or `flexseq()`): no change.

To insert one composite object (for example, a vector or a list) as a single element, wrap it in `list(...)`.

This operation is persistent: `x` is not modified.

Value

Updated sequence with inserted values.

Examples

```
x <- flexseq("a", "b", "c", "d")
insert_at(x, 3, c("x", "y"))
insert_at(x, 1, "start")
insert_at(x, length(x) + 1, "end")

# Insert one vector as a single element
insert_at(x, 3, list(c("u", "v")))
```

`interval_index` *Construct an Interval Index*

Description

Convenience constructor from `...`, `start`, and `end`.

Usage

```
interval_index(..., start, end, default_query_bounds = "[)")
```

Arguments

`...`	Elements to add.
`start`	Start endpoints matching `....`
`end`	End endpoints matching `....`
`default_query_bounds`	Boundary convention used as the default for query operations on this index: one of `"[)"`, `"[]"`, `"()"`, `"(]"`. Per-query `peek_*` / `pop_*` calls may override via their own `bounds` argument.

Details

Empty construction is supported: `interval_index()` returns an empty index.

Output is ordered by interval `start`.

Value

An `interval_index`.

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 2, 2), end = c(3, 2, 4))
ix

interval_index()
```

length.flexseq *Sequence Length*

Description

Sequence Length

Usage

```
## S3 method for class 'flexseq'
length(x)
```

Arguments

x	A flexseq.

Details

Uses cached size metadata and runs in O(1).

Value

Number of elements in the sequence.

Examples

```
x <- flexseq("a", "b")
length(x)

length(flexseq())
```

length.interval_index *Interval Index Length*

Description

Interval Index Length

Usage

```
## S3 method for class 'interval_index'
length(x)
```

Arguments

x	An interval_index.

Details

Uses cached size metadata and runs in O(1).

Value

Number of indexed intervals.

Examples

```
ix <- interval_index("a", "b", start = c(1, 3), end = c(2, 5))
length(ix)

length(interval_index())
```

---

length.ordered_sequence *Ordered Sequence Length*

Description

Ordered Sequence Length

Usage

```
## S3 method for class 'ordered_sequence'
length(x)
```

Arguments

x	An ordered_sequence.

Details

Uses cached size metadata and runs in O(1).

Value

Integer length.

Examples

```
x <- ordered_sequence("a", "b", keys = c(2, 1))
length(x)

length(ordered_sequence())
```

length.priority_queue *Priority Queue Length*

Description

Priority Queue Length

Usage

```
## S3 method for class 'priority_queue'
length(x)
```

Arguments

x	A priority_queue.

Details

Uses cached size metadata and runs in O(1).

Value

Integer length.

Examples

```
q <- priority_queue("a", "b", priorities = c(2, 1))
length(q)

length(priority_queue())
```

locate_by_predicate *Locate First Predicate Match*

Description

Scans accumulated monoid values and returns the first element where predicate becomes TRUE, without rebuilding split trees.

Usage

```
locate_by_predicate(
  t,
  predicate,
  monoid_name,
  accumulator = NULL,
  include_metadata = FALSE
)
```

Arguments

t	A flexseq (or subclass).
predicate	Function applied to accumulated monoid values.
monoid_name	Name of the monoid used for scanning.
accumulator	Optional starting accumulator value.
include_metadata	Logical; include scan metadata.

Details

This is the read-only analogue of split_around_by_predicate().

As with split helpers, a common setup is a custom monoid created with measure_monoid() and attached via add_monoids().

value is the matched leaf entry for the input structure:

- flexseq: stored user element.
- ordered_sequence: list(value, key).
- priority_queue: list(value, priority).
- interval_index: list(value, start, end).

Value

If include_metadata = FALSE, a list with:

- found: logical flag.
- value: matched element when found, otherwise NULL.

If include_metadata = TRUE, adds metadata with:

- left_measure
- hit_measure
- right_measure
- index

Examples

```
x <- flexseq("a", "b", "c", "d")
size_monoid <- measure_monoid(`+`, 0L, function(e) 1L)
x2 <- add_monoids(x, list(size = size_monoid))

locate_by_predicate(x2, function(v) v >= 3L, "size")
locate_by_predicate(x2, function(v) v >= 3L, "size", include_metadata = TRUE)
```

loop *Iterate over an iterator (re-exported from coro)*

Description

Re-exported `coro::loop()`. Enables `for`-loop-style iteration over `immutables` structures without needing to load `coro` separately.

Usage

```
loop(loop)
```

Arguments

`loop`	A `for` loop expression.

Value

`NULL`, invisibly. Called for side effects.

See Also

`coro::loop()` for full documentation.

Examples

```
s <- flexseq(1, 2, 3)
loop(for (x in s) print(x))
```

lower_bound *Find First Element with Key >= Query*

Description

Find First Element with Key >= Query

Usage

```
lower_bound(x, key)
```

Arguments

`x`	An `ordered_sequence`.
`key`	Query key.

Details

`lower_bound()` finds the first element with key `>= key`. This includes an exact key match when present, which is useful for starting equality or inclusive range scans.

Value

A list with fields:

- `found`: logical flag.
- `index`: one-based position of the first match, or `NULL`.
- `value`: matched element, or `NULL`.
- `key`: matched key, or `NULL`.

See Also

`upper_bound()`

Examples

```
x <- ordered_sequence("a", "b", "c", keys = c(1, 2, 2))
lower_bound(x, 2)
lower_bound(x, 10)
```

`max_endpoint` *Maximum Right Endpoint*

Description

Returns the largest right endpoint (`end`) currently present.

Usage

```
max_endpoint(x)
```

Arguments

`x`	An `interval_index`.

Details

Uses cached `.ivx_max_end` monoid state.

Value

Maximum right endpoint, or `NULL` when `x` is empty.

Examples

```
ix <- interval_index("a", "b", start = c(3, 1), end = c(4, 2))
max_endpoint(ix)
max_endpoint(interval_index())
```

`max_key` *Maximum Key Value*

Description

Returns the largest key currently present in the ordered sequence.

Usage

```
max_key(x)
```

Arguments

`x`	An `ordered_sequence`.

Details

Uses cached `.oms_max_key` monoid state.

Value

Maximum key, or `NULL` when `x` is empty.

Examples

```
x <- ordered_sequence("a", "b", keys = c(2, 1))
max_key(x)
max_key(ordered_sequence())
```

`max_priority` *Maximum Priority Value*

Description

Returns the current maximum priority scalar in the queue.

Usage

```
max_priority(x)
```

Arguments

x	A priority_queue.

Details

Uses cached .pq_max monoid state.

Value

Maximum priority value, or NULL when x is empty.

Examples

```
q <- priority_queue("a", "b", priorities = c(2, 1))
max_priority(q)
max_priority(priority_queue())
```

---

measure_monoid *Construct a Measure Monoid Specification*

Description

Defines how element-level values are measured and combined as accumulated tree metadata.

Usage

```
measure_monoid(f, i, measure)
```

Arguments

f	Associative binary function over measure values.
i	Identity value for f.
measure	Function mapping one element to its measure value.

Details

A measure monoid has three parts:

- measure(entry): maps each stored leaf entry to a measure value.
- f(left, right): combines two measure values.
- i: identity value for f.

Requirements:

- f should be associative.
- i should satisfy f(i, x) == x and f(x, i) == x.
- measure() outputs must be compatible with f and i.

Developer APIs are leaf-entry oriented:

- `flexseq`: entry is the stored user element.
- `ordered_sequence`: entry is `list(value, key)`.
- `priority_queue`: entry is `list(value, priority)`.
- `interval_index`: entry is `list(value, start, end)`.

`measure_monoid()` only constructs the specification; it becomes active after being attached to a structure via `add_monoids()`.

Value

An object of class `measure_monoid`.

Examples

```
sum_m <- measure_monoid(`+`, 0, as.numeric)
x <- as_flexseq(1:5)
x2 <- add_monoids(x, list(sum = sum_m))
attr(x2, "measures")$sum
split_around_by_predicate(x2, function(v) v >= 6, "sum")

# Count elements
count_m <- measure_monoid(`+`, 0L, function(el) 1L)
x3 <- add_monoids(x, list(count = count_m))
attr(x3, "measures")$count

# Character-width accumulation
width_m <- measure_monoid(`+`, 0L, function(el) nchar(as.character(el)))
s <- as_flexseq(c("aa", "b", "cccc"))
s2 <- add_monoids(s, list(width = width_m))
attr(s2, "measures")$width
```

`merge.flexseq` *Merge Two Sequences*

Description

Returns a new `flexseq` containing all elements of `x` followed by all elements of `y`. Thin wrapper over `c()` for API uniformity across the package's merge methods; `c(x, y)` and `merge(x, y)` are equivalent for `flexseq`.

Usage

```
## S3 method for class 'flexseq'
merge(x, y, ...)
```

Arguments

x	A flexseq.
y	A flexseq.
...	Unused.

Details

For ordered types (ordered_sequence, interval_index), merge() performs a proper sorted merge respecting keys/intervals — see merge.ordered_sequence() and merge.interval_index(). For priority_queue, see merge.priority_queue().

Value

A new flexseq.

Examples

```
x <- flexseq("a", "b")
y <- flexseq("c", "d")
merge(x, y)
```

---

merge.interval_index *Merge Two Interval Indices*

Description

Returns a new interval_index containing every entry from both inputs, preserving start-position order. On tied start positions, x's entries precede y's (left-biased FIFO).

Usage

```
## S3 method for class 'interval_index'
merge(x, y, ...)
```

Arguments

x	An interval_index.
y	An interval_index.
...	Unused.

Details

The merge runs in O(m + n) via a zipper-style traversal on interval starts, with a fast path to O(log(min(m, n))) when the start ranges are disjoint.

Both indices must share the same endpoint type and the same `bounds` convention (e.g. `"[)"` half-open vs. `"[]"` closed), and the same monoid set. Mismatches error.

The reserved monoids `.ivx_min_end` / `.ivx_max_end` recompute automatically on the merged tree, so `min_endpoint()` / `max_endpoint()` and interval-relation queries work immediately on the result.

Both inputs are left unmodified.

Value

A new `interval_index` of size `length(x) + length(y)`.

Examples

```
a <- interval_index("A1", "A2", start = c(1, 5), end = c(4, 8))
b <- interval_index("B1", "B2", start = c(3, 7), end = c(6, 10))
m <- merge(a, b)
as.list(m)
```

`merge.ordered_sequence`

Merge Two Ordered Sequences

Description

Returns a new `ordered_sequence` containing every entry from both inputs, preserving key order. On duplicate keys, `x`'s entries precede `y`'s (left-biased FIFO).

Usage

```
## S3 method for class 'ordered_sequence'
merge(x, y, ...)
```

Arguments

`x`	An `ordered_sequence`.
`y`	An `ordered_sequence`.
`...`	Unused.

Details

The merge runs in O(m + n) via a zipper-style traversal, with a fast path to O(log(min(m, n))) when the key ranges are disjoint (all of x's keys <= all of y's keys, or vice versa with a strict < to keep left-biased FIFO intact on equal boundary keys).

Both sequences must share the same key type and the same monoid set; mismatches error rather than being silently harmonized. Merging an empty sequence with a non-empty sequence returns the non-empty one unchanged.

Both inputs are left unmodified.

Value

A new `ordered_sequence` of size `length(x) + length(y)`.

Examples

```
a <- ordered_sequence("a1", "a2", "a3", keys = c(1, 3, 5))
b <- ordered_sequence("b1", "b2", "b3", keys = c(2, 3, 6))
m <- merge(a, b)
as.list(m)
# At the tied key 3, "a2" precedes "b2".
```

`merge.priority_queue` *Merge Two Priority Queues*

Description

Returns a new `priority_queue` containing every entry from both inputs, preserving each queue's internal insertion order (entries of `x` come first, then entries of `y`).

Usage

```
## S3 method for class 'priority_queue'
merge(x, y, ...)
```

Arguments

`x`	A `priority_queue`.
`y`	A `priority_queue`.
`...`	Unused.

Details

The cached `.pq_min` / `.pq_max` monoids recompute automatically on the merged tree, so `peek_min()` / `peek_max()` reflect the combined extremum immediately.

Both queues must share the same priority type and the same monoid set; mismatches error rather than being silently harmonized. Merging an empty queue with a non-empty queue returns the non-empty queue unchanged.

Both inputs are left unmodified.

Value

A new `priority_queue` of size `length(x) + length(y)`.

Examples

```
a <- priority_queue("x", "y", priorities = c(5, 1))
b <- priority_queue("z", priorities = 3)
m <- merge(a, b)
peek_min(m)
length(m)
```

`min_endpoint` *Minimum Left Endpoint*

Description

Returns the smallest left endpoint (`start`) currently present.

Usage

```
min_endpoint(x)
```

Arguments

`x`	An `interval_index`.

Details

Because intervals are stored in start order, this reads the first entry's `start`.

Value

Minimum left endpoint, or `NULL` when `x` is empty.

Examples

```
ix <- interval_index("a", "b", start = c(3, 1), end = c(4, 2))
min_endpoint(ix)
min_endpoint(interval_index())
```

min_key *Minimum Key Value*

Description

Returns the smallest key currently present in the ordered sequence.

Usage

```
min_key(x)
```

Arguments

x	An ordered_sequence.

Details

This follows sequence key order directly.

Value

Minimum key, or NULL when x is empty.

Examples

```
x <- ordered_sequence("a", "b", keys = c(2, 1))
min_key(x)
min_key(ordered_sequence())
```

min_priority *Minimum Priority Value*

Description

Returns the current minimum priority scalar in the queue.

Usage

```
min_priority(x)
```

Arguments

x	A priority_queue.

Details

Uses cached .pq_min monoid state.

Value

Minimum priority value, or `NULL` when `x` is empty.

Examples

```
q <- priority_queue("a", "b", priorities = c(2, 1))
min_priority(q)
min_priority(priority_queue())
```

`ordered_sequence` *Construct an Ordered Sequence*

Description

Convenience constructor from `...` and matching `keys`.

Usage

```
ordered_sequence(..., keys)
```

Arguments

`...`	Elements to add.
`keys`	Key values with the same length as `...`.

Details

Empty construction is supported: `ordered_sequence()` returns an empty ordered sequence.

Output is always sorted by key, with stable order across duplicate keys.

Value

An `ordered_sequence`.

Examples

```
xs <- ordered_sequence("bb", "a", "ccc", keys = c(2, 1, 3))
xs
lower_bound(xs, 2)

num_by_chr <- ordered_sequence(20, 10, 30, keys = c("b", "a", "c"))
num_by_chr

ordered_sequence()
```

peek_all_containing *Peek All Intervals Containing a Query Interval*

Description

Peek All Intervals Containing a Query Interval

Usage

```
peek_all_containing(x, start, end, bounds = NULL)
```

Arguments

x	An interval_index.
start	Query interval start.
end	Query interval end.
bounds	Optional boundary override. One of "[)", "[]", "()", "(]".

Details

The returned interval_index can be inspected with as.list().

Value

An interval_index slice of all matches (possibly empty).

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 2, 4), end = c(6, 5, 7))
as.list(peek_all_containing(ix, 2, 4))
```

peek_all_key *Peek All Elements for One Key*

Description

Returns all elements whose key equals key.

Usage

```
peek_all_key(x, key)
```

Arguments

x	An ordered_sequence.
key	Query key.

Details

The returned ordered_sequence can be inspected with as.list().

Value

An ordered_sequence containing all matches; empty on miss.

Examples

```
x <- ordered_sequence("a", "b", "c", keys = c(1, 2, 2))
out <- peek_all_key(x, 2)
as.list(out)
```

---

peek_all_max *Peek All Maximum-Priority Elements*

Description

Returns the full maximum-priority tie run as a priority_queue.

Usage

```
peek_all_max(x)
```

Arguments

x	A priority_queue.

Details

The return is another priority_queue(), use as.list() to convert the result to a standard R list.

Value

A priority_queue containing all maximum-priority elements in stable queue order. Returns an empty queue when x is empty.

Examples

```
x <- priority_queue("a", "b", "c", priorities = c(2, 3, 3))
peek_all_max(x)
```

peek_all_min *Peek All Minimum-Priority Elements*

Description

Returns the full minimum-priority tie run as a priority_queue.

Usage

```
peek_all_min(x)
```

Arguments

x	A priority_queue.

Details

The return is another priority_queue(), use as.list() to convert the result to a standard R list.

Value

A priority_queue containing all minimum-priority elements in stable queue order. Returns an empty queue when x is empty.

Examples

```
x <- priority_queue("a", "b", "c", priorities = c(2, 1, 1))
peek_all_min(x)
```

peek_all_overlaps *Peek All Intervals Overlapping a Query Interval*

Description

Peek All Intervals Overlapping a Query Interval

Usage

```
peek_all_overlaps(x, start, end, bounds = NULL)
```

Arguments

x	An interval_index.
start	Query interval start.
end	Query interval end.
bounds	Optional boundary override. One of "[)", "[]", "()", "(]".

Details

The returned interval_index can be inspected with as.list().

Value

An interval_index slice of all matches (possibly empty).

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 3, 5), end = c(2, 4, 6))
as.list(peek_all_overlaps(ix, 2, 5))
```

---

peek_all_point *Peek All Intervals Matching a Point*

Description

Peek All Intervals Matching a Point

Usage

```
peek_all_point(
  x,
  point,
  bounds = NULL,
  match_at = c("interval", "start", "end", "either")
)
```

Arguments

x	An interval_index.
point	Query point.
bounds	Optional boundary override. One of "[)", "[]", "()", "(]". Ignored when match_at is not "interval".
match_at	How the query point is matched against each entry. One of "interval" (default; containment under bounds), "start", "end", or "either". See peek_point() for details.

Details

The returned interval_index can be inspected with as.list().

Value

An interval_index slice of all matches (possibly empty).

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 2, 4), end = c(3, 2, 5))
as.list(peek_all_point(ix, 2))

# Entries ending at 3
as.list(peek_all_point(ix, 3, match_at = "end"))
```

---

peek_all_within *Peek All Intervals Within a Query Interval*

Description

Peek All Intervals Within a Query Interval

Usage

```
peek_all_within(x, start, end, bounds = NULL)
```

Arguments

x	An interval_index.
start	Query interval start.
end	Query interval end.
bounds	Optional boundary override. One of "[)", "[]", "()", "(]".

Details

The returned interval_index can be inspected with as.list().

Value

An interval_index slice of all matches (possibly empty).

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 2, 4), end = c(6, 3, 5))
as.list(peek_all_within(ix, 1, 4))
```

peek_at *Peek at an Element by Position*

Description

Returns the element at a one-based index without modifying the sequence.

Usage

```
peek_at(x, index)
```

Arguments

x	A flexseq.
index	One-based position to read.

Details

Positive integer indices beyond length(x) return NULL. Invalid indices (NA, non-integer, <= 0, or length not equal to 1) error.

Value

Element at index, or NULL when index is out of bounds.

Examples

```
x <- flexseq("a", "b", "c")
peek_at(x, 2)
peek_at(x, 10)

try(peek_at(x, 0))
```

peek_back *Peek at the Back Element*

Description

Returns the last element without modifying the sequence.

Usage

```
peek_back(x)
```

Arguments

x	A flexseq.

Details

Returns the payload element without modifying `x`.

Value

Last element, or `NULL` when `x` is empty.

Examples

```
x <- flexseq("a", "b", "c")
peek_back(x)

peek_back(flexseq())
```

---

peek_containing *Peek First Interval Containing a Query Interval*

Description

Peek First Interval Containing a Query Interval

Usage

```
peek_containing(x, start, end, bounds = NULL)
```

Arguments

`x`	An `interval_index`.
`start`	Query interval start.
`end`	Query interval end.
`bounds`	Optional boundary override. One of `"[)"`, `"[]"`, `"()"`, `"(]"`.

Details

Returns the first match in canonical interval order. Use `peek_all_containing()` to retrieve all matches as an `interval_index` slice.

Value

The payload value from the first match, or `NULL` on no match.

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 2, 4), end = c(5, 3, 6))
peek_containing(ix, 2, 3)
```

peek_front *Peek at the Front Element*

Description

Returns the first element without modifying the sequence.

Usage

```
peek_front(x)
```

Arguments

x	A flexseq.

Details

Returns the payload element without modifying x.

Value

First element, or NULL when x is empty.

Examples

```
x <- flexseq("a", "b", "c")
peek_front(x)

peek_front(flexseq())
```

peek_key *Peek First Element for One Key*

Description

Returns the first element whose key equals key.

Usage

```
peek_key(x, key)
```

Arguments

x	An ordered_sequence.
key	Query key.

Details

For duplicate keys, this returns the first element in stable sequence order.

Value

Matched element, or NULL when no matching key exists.

Examples

```
x <- ordered_sequence("a", "b", "c", keys = c(1, 2, 2))
peek_key(x, 2)
peek_key(x, 10)
```

---

peek_max *Peek Maximum-Priority Element*

Description

Returns the element at the maximum priority without modifying the queue.

Usage

```
peek_max(x)
```

Arguments

x	A priority_queue.

Details

Ties are stable: when multiple elements share maximum priority, this returns the earliest element in queue order.

Value

Element at maximum priority, or NULL when x is empty.

Examples

```
x <- priority_queue("a", "b", "c", priorities = c(2, 3, 3))
peek_max(x)
peek_max(priority_queue())
```

peek_min *Peek Minimum-Priority Element*

Description

Returns the element at the minimum priority without modifying the queue.

Usage

```
peek_min(x)
```

Arguments

x	A priority_queue.

Details

Ties are stable: when multiple elements share minimum priority, this returns the earliest element in queue order.

Value

Element at minimum priority, or NULL when x is empty.

Examples

```
x <- priority_queue("a", "b", "c", priorities = c(2, 1, 1))
peek_min(x)
peek_min(priority_queue())
```

peek_overlaps *Peek First Interval Overlapping a Query Interval*

Description

Peek First Interval Overlapping a Query Interval

Usage

```
peek_overlaps(x, start, end, bounds = NULL)
```

Arguments

x	An interval_index.
start	Query interval start.
end	Query interval end.
bounds	Optional boundary override. One of "[)", "[]", "()", "(]".

Details

Returns the first match in canonical interval order. Use peek_all_overlaps() to retrieve all matches as an interval_index slice.

Value

The payload value from the first match, or NULL on no match.

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 3, 5), end = c(2, 4, 6))
peek_overlaps(ix, 2, 3)

# Boundary override at touching endpoints
edge <- interval_index("a", start = 1, end = 3, default_query_bounds = "[)")
peek_overlaps(edge, 3, 4)                # default "[)": no endpoint overlap
peek_overlaps(edge, 3, 4, bounds = "[]") # closed bounds: endpoint overlaps
```

---

peek_point *Peek First Interval Matching a Point*

Description

Peek First Interval Matching a Point

Usage

```
peek_point(
  x,
  point,
  bounds = NULL,
  match_at = c("interval", "start", "end", "either")
)
```

Arguments

x	An interval_index.
point	Query point.
bounds	Optional boundary override. One of "[)", "[]", "()", "(]". Ignored when match_at is not "interval".
match_at	How the query point is matched against each entry. One of "interval" (default; entry interval contains point, under bounds), "start" (entry's start coordinate equals point), "end" (entry's end equals point), or "either" (start or end equals point). The three coordinate-equality modes are structural and ignore bounds.

Details

Returns the first match in canonical interval order. Use peek_all_point() to retrieve all matches as an interval_index slice.

Value

The payload value from the first match, or NULL on no match.

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 2, 4), end = c(3, 2, 5))
peek_point(ix, 2)

# Boundary override at an endpoint
edge <- interval_index("a", start = 1, end = 3, default_query_bounds = "[)")
peek_point(edge, 3)                # default "[)": no match at right endpoint
peek_point(edge, 3, bounds = "[]") # closed bounds: endpoint matches

# Coordinate-equality modes (bounds irrelevant)
peek_point(ix, 2, match_at = "start")  # entry starting at 2
peek_point(ix, 3, match_at = "end")    # entry ending at 3
```

---

peek_within *Peek First Interval Within a Query Interval*

Description

Peek First Interval Within a Query Interval

Usage

```
peek_within(x, start, end, bounds = NULL)
```

Arguments

x	An interval_index.
start	Query interval start.
end	Query interval end.
bounds	Optional boundary override. One of "[)", "[]", "()", "(]".

Details

Returns the first match in canonical interval order. Use peek_all_within() to retrieve all matches as an interval_index slice.

Value

The payload value from the first match, or NULL on no match.

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 2, 4), end = c(6, 3, 5))
peek_within(ix, 1, 4)
```

`plot_structure` *Plot the Internal Finger-Tree Structure*

Description

Developer-facing visualizer for the finger-tree backing any immutables structure (`flexseq`, `ordered_sequence`, `priority_queue`, `interval_index`). The S3 `plot()` methods on those classes forward to this function, but `plot_structure()` is also callable directly and gives access to the full `node_label` API for custom label formatting. Requires the `igraph` package (listed in `Suggests`).

Usage

```
plot_structure(
  t1,
  vertex.size = 15,
  vertex.shape = "rounded_rect",
  edge.width = 1,
  label_edges = FALSE,
  title = NULL,
  node_label = "value",
  label.cex = NULL,
  asp = NA,
  legend = TRUE,
  ...
)
```

Arguments

`t1`	A finger-tree-backed immutables structure.
`vertex.size`	Passed to `igraph::plot.igraph`. Ignored by the default `"rounded_rect"` shape, which sizes each box to its own label text.
`vertex.shape`	Vertex shape. Defaults to `"rounded_rect"`, a custom shape registered on first call that auto-sizes every box to fit its label (so multi-line labels render cleanly without tuning `vertex.size`). Any built-in igraph shape (`"circle"`, `"rectangle"`, `"none"`, ...) also works and is passed through unchanged.
`edge.width`	Edge width passed to `igraph::plot.igraph`.
`label_edges`	If `TRUE`, draw the child slot name on each edge (`"prefix"`/`"middle"`/`"suffix"` for `Deep` nodes, or the numeric position for `Digit`/`Node2`/`Node3`).
`title`	Optional plot title.

`node_label` Either one of the preset modes `"value"` (default; payload for elements, blank for structural), `"type"` (node class name), `"both"` (type + value on separate lines), `"none"`, or a user-supplied function `function(node)` returning a single character string per vertex. The `node` argument is a list with fields:

`id` Internal graph vertex id (string).

`type` Node class: `"Element"`, `"Digit"`, `"Deep"`, `"Node2"`, `"Node3"`, `"Single"`, or `"Empty"`.

`label` The default label string.

`measures` Named list of accumulated monoid values for the subtree rooted at this node. For structural nodes these are the cached values; for element leaves, each entry is the monoid's `measure()` applied to the leaf entry. Keys include built-ins (`.size`, `.named_count`, and any structure-specific ones like `.pq_min`) plus any custom name added via `add_monoids()`.

`element` For element nodes, the raw leaf entry. Shape depends on the structure type; see `measure_monoid()` for the entry contract. `NULL` for structural nodes.

Measure values are exposed as-is, including list-valued measures (e.g. the built-in `.pq_min` is `list(has, priority)`).

`label.cex` Numeric scalar controlling label text size (passed as `vertex.label.cex` to igraph). If `NULL` (default), scales automatically from ~1.0 on small trees down to ~0.55 on dense ones so the auto-sized `"rounded_rect"` boxes don't collide. Override with a fixed value for finer control.

`asp` Plot aspect ratio (physical y-unit / physical x-unit). Default `NA` lets the tree fill the device without aspect constraint. Set to a numeric (e.g. `0.4`) to enforce horizontal stretching, or `1` for a square plot region.

`legend` If `TRUE` (default), draws a horizontal legend below the tree mapping node-type names to their fill colors, restricted to types actually present.

`...` Additional arguments passed to `igraph::plot.igraph` (e.g. `vertex.label.color`, `vertex.frame.color`).

Value

Invoked for its side effect (draws to the active graphics device). Returns `NULL` invisibly.

See Also

`measure_monoid()`, `add_monoids()`

Examples

```
if (requireNamespace("igraph", quietly = TRUE)) {

t <- as_flexseq(letters[1:8])
plot_structure(t, title = "Finger tree")

# Label every node with its subtree size (leaves contribute 1).
plot_structure(as_flexseq(1:10), node_label = function(node) {
```

```
  paste0(node$type, "\n.size=", node$measures$.size)
})

# Custom monoid: subtree sum of numeric payloads. Structural nodes show
# the accumulated total; leaves show their own contribution.
sum_monoid <- measure_monoid(`+`, 0, function(el) el)
xs <- add_monoids(as_flexseq(c(3, 1, 4, 1, 5, 9, 2, 6)),
                  list(sum = sum_monoid))
plot_structure(xs, node_label = function(node) {
  if(node$type == "Element") sprintf("%g\nsum=%g", node$element, node$measures$sum)
  else sprintf("%s\nsum=%g", node$type, node$measures$sum)
})

# List-valued built-in measure: priority_queue's .pq_min tracks the min
# priority seen in a subtree as list(has, priority). Unpack in the label.
pq <- priority_queue("task-a", "task-b", "task-c",
                     priorities = c(5, 1, 3))
plot_structure(pq, node_label = function(node) {
  m <- node$measures$.pq_min
  if(node$type == "Element") {
    sprintf("%s\np=%g", node$element$value, node$element$priority)
  } else if(isTRUE(m$has)) {
    sprintf("%s\nmin=%g", node$type, m$priority)
  } else {
    node$type
  }
})

}
```

---

pop_all_containing *Pop All Containing Intervals*

Description

Pop All Containing Intervals

Usage

```
pop_all_containing(x, start, end, bounds = NULL)
```

Arguments

x	An interval_index.
start	Query interval start.
end	Query interval end.
bounds	Optional boundary override. One of "[)", "[]", "()", "(]".

Details

Use `as.list()` to convert `elements` to a standard R list.

Value

A list with `elements` and `remaining`, both `interval_index` objects.

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 2, 4), end = c(6, 5, 7))
out <- pop_all_containing(ix, 2, 4)
as.list(out$elements)
```

---

pop_all_key *Pop All Elements for One Key*

Description

Removes and returns all elements whose key equals `key`.

Usage

```
pop_all_key(x, key)
```

Arguments

`x`	An `ordered_sequence`.
`key`	Query key.

Details

Use `as.list()` to convert `elements` to a standard R list.

Value

A list with fields:

- `elements`: `ordered_sequence` of removed matches.
- `remaining`: `ordered_sequence` after removal.

Examples

```
x <- ordered_sequence("a", "b", "c", keys = c(1, 2, 2))
out <- pop_all_key(x, 2)
as.list(out$elements)
out$remaining
```

pop_all_max *Pop All Maximum-Priority Elements*

Description

Removes the full maximum-priority tie run.

Usage

```
pop_all_max(x)
```

Arguments

x	A priority_queue.

Details

The return elements is another priority_queue(), use as.list() to convert the result to a standard R list.

Value

A list with fields:

- elements: priority_queue of removed maximum-priority elements.
- remaining: queue after removal.

Examples

```
x <- priority_queue("a", "b", "c", priorities = c(2, 3, 3))
out <- pop_all_max(x)
out$elements
out$remaining
```

pop_all_min *Pop All Minimum-Priority Elements*

Description

Removes the full minimum-priority tie run.

Usage

```
pop_all_min(x)
```

Arguments

x	A priority_queue.

Details

The return elements is another priority_queue(), use as.list() to convert the result to a standard R list.

Value

A list with fields:

- elements: priority_queue of removed minimum-priority elements.
- remaining: queue after removal.

Examples

```
x <- priority_queue("a", "b", "c", priorities = c(2, 1, 1))
out <- pop_all_min(x)
out$elements
out$remaining
```

---

pop_all_overlaps *Pop All Overlapping Intervals*

Description

Pop All Overlapping Intervals

Usage

```
pop_all_overlaps(x, start, end, bounds = NULL)
```

Arguments

x	An interval_index.
start	Query interval start.
end	Query interval end.
bounds	Optional boundary override. One of "[)", "[]", "()", "(]".

Details

Use as.list() to convert elements to a standard R list.

Value

A list with elements and remaining, both interval_index objects.

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 3, 5), end = c(2, 4, 6))
out <- pop_all_overlaps(ix, 2, 5)
as.list(out$elements)
```

---

pop_all_point *Pop All Intervals Matching a Point*

Description

Pop All Intervals Matching a Point

Usage

```
pop_all_point(
  x,
  point,
  bounds = NULL,
  match_at = c("interval", "start", "end", "either")
)
```

Arguments

x	An interval_index.
point	Query point.
bounds	Optional boundary override. One of "[)", "[]", "()", "(]". Ignored when match_at is not "interval".
match_at	How the query point is matched against each entry. One of "interval" (default; containment under bounds), "start", "end", or "either". See peek_point() for details. The "end" mode is the standard primitive for retiring active intervals in a sweep-line.

Details

Use as.list() to convert elements to a standard R list.

Value

A list with elements and remaining, both interval_index objects.

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 2, 4), end = c(3, 2, 5))
out <- pop_all_point(ix, 2)
as.list(out$elements)

# Sweep-line retirement: remove everything ending at x = 3
retired <- pop_all_point(ix, 3, match_at = "end")
as.list(retired$elements)
as.list(retired$remaining)
```

`pop_all_within` *Pop All Intervals Within a Query Interval*

Description

Pop All Intervals Within a Query Interval

Usage

```
pop_all_within(x, start, end, bounds = NULL)
```

Arguments

`x`	An `interval_index`.
`start`	Query interval start.
`end`	Query interval end.
`bounds`	Optional boundary override. One of `"[)"`, `"[]"`, `"()"`, `"(]"`.

Details

Use `as.list()` to convert `elements` to a standard R list.

Value

A list with `elements` and `remaining`, both `interval_index` objects.

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 2, 4), end = c(6, 3, 5))
out <- pop_all_within(ix, 1, 4)
as.list(out$elements)
```

pop_at *Pop an Element by Position*

Description

Returns the selected element and the remaining sequence.

Usage

```
pop_at(x, index)
```

Arguments

x	A flexseq.
index	One-based position to remove.

Details

This operation is persistent: x is not modified.

Positive integer indices beyond length(x) return a non-throwing miss object with value = NULL and remaining = x. Invalid indices (NA, non-integer, <= 0, or length not equal to 1) error.

Value

A list with fields:

- value: the element at index, or NULL when index is out of bounds.
- remaining: the sequence after removing the selected element.

Examples

```
x <- flexseq("a", "b", "c", "d")
out <- pop_at(x, 3)
out$value
out$remaining
x  # unchanged

pop_at(x, 10)
try(pop_at(x, 0))
```

pop_back *Pop the Back Element*

Description

Returns the last element and the remaining sequence.

Usage

```
pop_back(x)
```

Arguments

x	A flexseq.

Details

This operation is persistent: x is not modified.

On empty input, returns a non-throwing miss object with value = NULL and remaining = x.

Value

A list with fields:

- value: the last element, or NULL when x is empty.
- remaining: the sequence after removing the last element.

Examples

```
s <- flexseq("a", "b", "c")
out <- pop_back(s)
out$value
out$remaining
s  # unchanged

pop_back(flexseq())
```

pop_containing *Pop First Containing Interval*

Description

Pop First Containing Interval

Usage

```
pop_containing(x, start, end, bounds = NULL)
```

Arguments

x	An interval_index.
start	Query interval start.
end	Query interval end.
bounds	Optional boundary override. One of "[)", "[]", "()", "(]".

Details

Removes the first match in canonical interval order. On miss, returns a non-throwing miss object with remaining = x. Use pop_all_containing() to remove all matches.

Value

A list with element, start, end, and remaining. On miss: element, start, and end are NULL.

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 2, 4), end = c(6, 3, 7))
pop_containing(ix, 2, 4)
```

pop_front *Pop the Front Element*

Description

Returns the first element and the remaining sequence.

Usage

```
pop_front(x)
```

Arguments

x	A flexseq.

Details

This operation is persistent: `x` is not modified.

On empty input, returns a non-throwing miss object with `value = NULL` and `remaining = x`.

Value

A list with fields:

- `value`: the first element, or `NULL` when `x` is empty.
- `remaining`: the sequence after removing the first element.

Examples

```
s <- flexseq("a", "b", "c")
out <- pop_front(s)
out$value
out$remaining
s  # unchanged

pop_front(flexseq())
```

---

pop_key *Pop First Element for One Key*

Description

Removes and returns the first element whose key equals `key`.

Usage

```
pop_key(x, key)
```

Arguments

`x`	An `ordered_sequence`.
`key`	Query key.

Details

For duplicate keys, the first element in stable sequence order is removed.

Value

A list with fields:

- `value`: removed element, or `NULL` on miss.
- `key`: removed key, or `NULL` on miss.
- `remaining`: ordered sequence after removal.

Examples

```
x <- ordered_sequence("a", "b", "c", keys = c(1, 2, 2))
out <- pop_key(x, 2)
out$value
out$remaining
pop_key(x, 10)
```

---

pop_max *Pop Maximum-Priority Element*

Description

Removes one maximum-priority element and returns it with the remaining queue.

Usage

```
pop_max(x)
```

Arguments

x	A priority_queue.

Details

Ties are stable: when multiple elements share maximum priority, the earliest element in queue order is removed.

Value

A list with fields:

- value: removed element, or NULL when x is empty.
- priority: removed priority, or NULL when x is empty.
- remaining: queue after removal.

Examples

```
x <- priority_queue("a", "b", "c", priorities = c(2, 3, 3))
out <- pop_max(x)
out$value
out$priority
out$remaining
pop_max(priority_queue())
```

pop_min *Pop Minimum-Priority Element*

Description

Removes one minimum-priority element and returns it with the remaining queue.

Usage

```
pop_min(x)
```

Arguments

x	A priority_queue.

Details

Ties are stable: when multiple elements share minimum priority, the earliest element in queue order is removed.

Value

A list with fields:

- value: removed element, or NULL when x is empty.
- priority: removed priority, or NULL when x is empty.
- remaining: queue after removal.

Examples

```
x <- priority_queue("a", "b", "c", priorities = c(2, 1, 1))
out <- pop_min(x)
out$value
out$priority
out$remaining
pop_min(priority_queue())
```

pop_overlaps *Pop First Overlapping Interval*

Description

Pop First Overlapping Interval

Usage

```
pop_overlaps(x, start, end, bounds = NULL)
```

Arguments

x	An interval_index.
start	Query interval start.
end	Query interval end.
bounds	Optional boundary override. One of "[)", "[]", "()", "(]".

Details

Removes the first match in canonical interval order. On miss, returns a non-throwing miss object with remaining = x. Use pop_all_overlaps() to remove all matches.

Value

A list with element, start, end, and remaining. On miss: element, start, and end are NULL.

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 3, 5), end = c(2, 4, 6))
pop_overlaps(ix, 2, 3)
```

pop_point *Pop First Interval Matching a Point*

Description

Pop First Interval Matching a Point

Usage

```
pop_point(
  x,
  point,
  bounds = NULL,
  match_at = c("interval", "start", "end", "either")
)
```

Arguments

x	An interval_index.
point	Query point.
bounds	Optional boundary override. One of "[)", "[]", "()", "(]". Ignored when match_at is not "interval".
match_at	How the query point is matched against each entry. One of "interval" (default; containment under bounds), "start", "end", or "either". See peek_point() for details.

Details

Removes the first match in canonical interval order. On miss, returns a non-throwing miss object with remaining = x. Use pop_all_point() to remove all matches.

Value

A list with element, start, end, and remaining. On miss: element, start, and end are NULL.

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 2, 4), end = c(3, 2, 5))
pop_point(ix, 2)
```

---

pop_within *Pop First Interval Within a Query Interval*

Description

Pop First Interval Within a Query Interval

Usage

```
pop_within(x, start, end, bounds = NULL)
```

Arguments

x	An interval_index.
start	Query interval start.
end	Query interval end.
bounds	Optional boundary override. One of "[)", "[]", "()", "(]".

Details

Removes the first match in canonical interval order. On miss, returns a non-throwing miss object with remaining = x. Use pop_all_within() to remove all matches.

Value

A list with element, start, end, and remaining. On miss: element, start, and end are NULL.

Examples

```
ix <- interval_index("a", "b", "c", start = c(1, 2, 4), end = c(6, 3, 5))
pop_within(ix, 1, 4)
```

---

print.flexseq *Print a flexseq*

Description

Print a flexseq

Usage

```
## S3 method for class 'flexseq'
print(x, max_elements = 4L, show_custom_monoids = FALSE, ...)
```

Arguments

x	A flexseq.
max_elements	Maximum number of elements shown in preview (head + tail).
show_custom_monoids	Logical; show attached non-default monoids and their root cached measures.
...	Passed through to per-element print().

Value

The input x, returned invisibly. Called for its side effect of printing a formatted preview of the sequence to the console.

Examples

```
x <- as_flexseq(setNames(as.list(1:6), letters[1:6]))
print(x, max_elements = 4)

y <- as_flexseq(as.list(1:6))
print(y, max_elements = 3)
```

`print.interval_index` *Print an Interval Index Summary*

Description

Prints a compact summary with interval bounds and a head/tail preview of payload elements.

Usage

```
## S3 method for class 'interval_index'
print(x, max_elements = 4L, show_custom_monoids = FALSE, ...)
```

Arguments

`x`	An `interval_index`.
`max_elements`	Maximum number of elements shown in the preview.
`show_custom_monoids`	Logical; show attached non-default monoids and their root cached measures.
`...`	Passed through to per-element `print()`.

Value

Invisibly returns `x`.

Examples

```
ix <- interval_index(
  one = 1, two = 2, three = 3,
  start = c(20, 30, 10), end = c(25, 37, 24)
)
print(ix, max_elements = 4)
width_sum <- measure_monoid(
  `+`, 0, function(entry) as.numeric(entry$end - entry$start)
)
ix3 <- add_monoids(
  interval_index(1, 2, start = c(1, 3), end = c(2, 5)),
  list(width_sum = width_sum)
)
print(ix3, max_elements = 0, show_custom_monoids = TRUE)

ix2 <- interval_index(1, 2, 3, start = c(2, 4, 6), end = c(3, 5, 8), default_query_bounds = "[]")
print(ix2, max_elements = 3)

print(interval_index())
```

print.ordered_sequence *Print an Ordered Sequence Summary*

Description

Prints a compact summary and head/tail preview in key order.

Usage

```
## S3 method for class 'ordered_sequence'
print(x, max_elements = 4L, show_custom_monoids = FALSE, ...)
```

Arguments

x	An ordered_sequence.
max_elements	Maximum number of elements shown in the preview.
show_custom_monoids	Logical; show attached non-default monoids and their root cached measures.
...	Passed through to per-element print().

Value

Invisibly returns x.

Examples

```
xs <- ordered_sequence(one = "a", two = "b", three = "c", keys = c(20, 30, 10))
print(xs, max_elements = 4)
sum_key <- measure_monoid(`+`, 0, function(entry) as.numeric(entry$key))
ys2 <- add_monoids(ordered_sequence("a", "b", keys = c(2, 1)), list(sum_key = sum_key))
print(ys2, max_elements = 0, show_custom_monoids = TRUE)

ys <- ordered_sequence("x", "y", "z", keys = c(2, 1, 3))
print(ys, max_elements = 3)

print(ordered_sequence())
```

print.priority_queue *Print a Priority Queue Summary*

Description

Prints a compact summary with priority range and a head/tail preview.

Usage

```
## S3 method for class 'priority_queue'
print(x, max_elements = 4L, show_custom_monoids = FALSE, ...)
```

Arguments

x	A priority_queue.
max_elements	Maximum number of elements shown in the preview.
show_custom_monoids	Logical; show attached non-default monoids and their root cached measures.
...	Passed through to per-element print().

Value

Invisibly returns x.

Examples

```
q <- priority_queue(one = 1, two = 2, three = 3, priorities = c(20, 30, 10))
print(q, max_elements = 4)
sum_item <- measure_monoid(`+`, 0, function(entry) as.numeric(entry$value))
q3 <- add_monoids(priority_queue(1, 2, priorities = c(2, 1)), list(sum_item = sum_item))
print(q3, max_elements = 0, show_custom_monoids = TRUE)

q2 <- priority_queue(1, 2, 3, priorities = c(2, 1, 3))
print(q2, max_elements = 3)

print(priority_queue())
```

priority_queue *Construct a Priority Queue*

Description

Creates a priority_queue from elements in ... and matching priorities.

Usage

```
priority_queue(..., priorities)
```

Arguments

`...`	Elements to enqueue.
`priorities`	Priorities with the same length as `...`.

Details

Empty construction is supported: `priority_queue()` returns an empty queue.

If elements are named, names are preserved for name-based reads.

Queue operations are exposed through `insert()`, `peek_*()`, `pop_*()`, and `fapply()`.

Value

A `priority_queue`.

Examples

```
x <- priority_queue("a", "b", "c", priorities = c(2, 1, 2))
x
peek_min(x)

empty_q <- priority_queue()
peek_min(empty_q)
```

---

push_back *Push an Element to the Back*

Description

Returns a new sequence with `value` appended at the right end.

Usage

```
push_back(x, value)
```

Arguments

`x`	A `flexseq`.
`value`	Element to append.

Details

This operation is persistent: `x` is not modified.

Elements can be named, but only if all are uniquely named (no missing names).

Value

Updated `flexseq`.

Examples

```
s <- as_flexseq(letters[1:3])
s2 <- push_back(s, "d")
s2
s  # unchanged

n <- as_flexseq(list(two = 2, three = 3))
new_el <- 4
names(new_el) <- "four"
push_back(n, new_el)

# Named/unnamed mixes are rejected
try(push_back(n, 5))
```

---

push_front *Push an Element to the Front*

Description

Returns a new sequence with `value` prepended at the left end.

Usage

```
push_front(x, value)
```

Arguments

`x`	A `flexseq`.
`value`	Element to prepend.

Details

This operation is persistent: `x` is not modified.

Elements can be named, but only if all are uniquely named (no missing names).

Value

Updated `flexseq`.

Examples

```
s <- as_flexseq(letters[2:4])
s2 <- push_front(s, "a")
s2
s  # unchanged

n <- as_flexseq(list(two = 2, three = 3))
new_el <- 1
```

```
names(new_el) <- "one"
push_front(n, new_el)

# Named/unnamed mixes are rejected
try(push_front(n, 0))
```

`split_around_by_predicate` *Split Around First Predicate Match*

Description

Splits at the first point where a `predicate` function becomes `TRUE` while scanning the sequence.

Usage

```
split_around_by_predicate(t, predicate, monoid_name, accumulator = NULL)
```

Arguments

`t`	A `flexseq` (or subclass).
`predicate`	Function applied to accumulated monoid values.
`monoid_name`	Name of the monoid used for scanning.
`accumulator`	Optional starting accumulator value.

Details

This function generally requires the sequence be annotated with a `measure_monoid()`; see the examples and `measure_monoid()` for more information.

`value` is the matched leaf entry for the input structure:

- `flexseq`: stored user element.
- `ordered_sequence`: `list(value, key)`.
- `priority_queue`: `list(value, priority)`.
- `interval_index`: `list(value, start, end)`.

`left` and `right` preserve subclass when the input is a subclass of `flexseq`.

Value

A list with fields:

- `left`: elements before the split point.
- `value`: the matched element at the split point.
- `right`: elements after the split point.

Examples

```
x <- flexseq("a", "b", "c", "d")

# Each element e has measure 1; the accumulated measure for
# a set of elements is computed by the associative function `+`
# along with the identity value 0.
size_monoid <- measure_monoid(`+`, 0L, function(e) 1L)
x2 <- add_monoids(x, list(size = size_monoid))

# the first time the measure stored in the size monoid
# accumulates to greater than or equal to 3 is the 3rd
# element in sequence.
split_around_by_predicate(x2, function(v) v >= 3L, "size")

# Split at the first element
split_around_by_predicate(x2, function(v) v >= 1L, "size")

# Split at the last element
split_around_by_predicate(x2, function(v) v >= 4L, "size")
```

---

split_at *Split at a Position or Name*

Description

Splits by a single one-based position or a single element name.

Usage

```
split_at(x, at, pull_index = FALSE)
```

Arguments

x	A flexseq.
at	A single positive integer position or a single character name.
pull_index	Controls output shape: • FALSE (default): returns list(left, value, right). • TRUE: returns list(left, right).

Details

split_at(x, at, pull_index = FALSE) is a convenience wrapper around split_around_by_predicate() using positional scanning.

split_at(x, at, pull_index = TRUE) is the two-way variant using split_by_predicate().

Value

A split result with shape controlled by pull_index.

Examples

```
x <- flexseq("a", "b", "c", "d")
split_at(x, 3)
split_at(x, 3, pull_index = TRUE)

n <- flexseq(a = 1, b = 2, c = 3)
split_at(n, "b")
```

---

split_by_predicate *Split into Left and Right Parts*

Description

Splits at the first point where `predicate` becomes `TRUE` while scanning the sequence.

Usage

```
split_by_predicate(x, predicate, monoid_name)
```

Arguments

`x`	A `flexseq` (or subclass).
`predicate`	Function applied to accumulated monoid values.
`monoid_name`	Name of the monoid used for scanning.

Details

This is the two-way variant of `split_around_by_predicate()`.

As with `split_around_by_predicate()`, a common setup is a custom monoid created with `measure_monoid()` and attached via `add_monoids()`.

`left` and `right` preserve subclass when the input is a subclass of `flexseq`.

Value

A list with fields:

- `left`: elements before the split point.
- `right`: elements from the split point onward.

Examples

```
x <- flexseq("a", "b", "c", "d")
size_monoid <- measure_monoid(`+`, 0L, function(e) 1L)
x2 <- add_monoids(x, list(size = size_monoid))
split_by_predicate(x2, function(v) v >= 3L, "size")

# Split at the first element
split_by_predicate(x2, function(v) v >= 1L, "size")
```

`sub-.interval_index` *Indexing for Interval Indexes*

Description

Read indexing returns `interval_index` subsets while preserving interval/key order; out-of-order selectors are canonicalized with a warning. Replacement indexing is blocked.

Usage

```
## S3 method for class 'interval_index'
x[i, ...]

## S3 method for class 'interval_index'
x[[i, ...]]

## S3 replacement method for class 'interval_index'
x[i] <- value

## S3 replacement method for class 'interval_index'
x[[i]] <- value

## S3 method for class 'interval_index'
x$name

## S3 replacement method for class 'interval_index'
x$name <- value
```

Arguments

`x`	An `interval_index`.
`i`	Index input.
`...`	Unused.
`value`	Replacement value (unsupported).
`name`	Element name (for `$` and `$<-`).

Details

Read indexing preserves canonical interval order in returned subsets.

- Integer/character vectors are treated as selectors and canonicalized to interval-order output.
- Out-of-order selector vectors trigger a warning and are canonicalized.
- Duplicate selectors are rejected.
- `[[` and `$` return payload values.
- Replacement indexing (`[<-`, `[[<-`, `$<-`) is unsupported.

Value

Read methods return interval payload values/subsets; replacement forms always error.

For `$`: the matched payload element.

No return value; always errors because replacement indexing is unsupported.

Examples

```
ix <- interval_index(a = "A", b = "B", c = "C", start = c(1, 3, 5), end = c(2, 4, 6))

ix[c(3, 1)]         # warning; result returned in interval order
ix[c("c", "a")]     # warning; result returned in interval order
ix[c(TRUE, FALSE, TRUE)]
ix[["b"]]
ix$b

try(ix[c(2, 2)])
try(ix$b <- "updated")
```

---

sub-.ordered_sequence *Indexing for Ordered Sequences*

Description

Read indexing treats vectors as selectors and returns subsets in key order. Out-of-order selectors are canonicalized with a warning. Replacement indexing is blocked to prevent order-breaking writes.

Usage

```
## S3 method for class 'ordered_sequence'
x[i, ...]

## S3 method for class 'ordered_sequence'
x[[i, ...]]

## S3 replacement method for class 'ordered_sequence'
x[i] <- value

## S3 replacement method for class 'ordered_sequence'
x[[i]] <- value

## S3 method for class 'ordered_sequence'
x$name

## S3 replacement method for class 'ordered_sequence'
x$name <- value
```

Arguments

x	An ordered_sequence.
i	Index input.
...	Unused.
value	Replacement value (unsupported).
name	Element name (for $ and $<-).

Details

Vector selectors are treated as membership selectors, not output-order instructions.

- Integer/character vectors are normalized to unique positions and returned in canonical sequence order.
- Out-of-order selector vectors trigger a warning and are canonicalized.
- Duplicate selectors are rejected.
- Replacement indexing ([<-, [[<-, $<-) is unsupported.

Value

Read methods return ordered payload values/subsets; replacement forms always error.

Examples

```
x <- ordered_sequence(a = "A", b = "B", c = "C", keys = c(1, 2, 3))

x[c(3, 1)]          # warning; result returned in key order
x[c("c", "a")]      # warning; result returned in key order
x[c(TRUE, FALSE, TRUE)]
x[["b"]]
x$b

try(x[c(2, 2)])
try(x$b <- "updated")
```

---

sub-.priority_queue *Indexing for Priority Queues*

Description

Name-based indexing is supported for reads only. Positional indexing and all replacement indexing are intentionally blocked to preserve queue-first UX.

Usage

```
## S3 method for class 'priority_queue'
x[i, ...]

## S3 method for class 'priority_queue'
x[[i, ...]]

## S3 replacement method for class 'priority_queue'
x[i] <- value

## S3 replacement method for class 'priority_queue'
x[[i]] <- value

## S3 method for class 'priority_queue'
x$name

## S3 replacement method for class 'priority_queue'
x$name <- value
```

Arguments

x	A priority_queue.
i	Index input. For reads, must be a character name (scalar for [[).
...	Unused.
value	Replacement value (unsupported).
name	Element name (for $ and $<-).

Details

priority_queue supports name-based read indexing only.

- [: character vector of names, returns a priority_queue subset.
- [[and $: scalar name, return the payload value.
- Positional indexing and all replacement indexing forms are unsupported.

Value

For $/[[/[: queue payload values or queue subsets by name. Replacement forms always error.

Examples

```
q <- priority_queue(a = "task-a", b = "task-b", priorities = c(2, 1))

q["a"]
q[["b"]]
q$b

try(q[1])
try(q$a <- "updated")
```

`unlist.flexseq` *Coerce a Sequence to an Atomic Vector*

Description

Convenience wrapper around `base::unlist()` over `as.list()`.

Usage

```
## S3 method for class 'flexseq'
unlist(x, recursive = TRUE, use.names = TRUE)
```

Arguments

`x`	A `flexseq`.
`recursive`	Passed through to `base::unlist()`.
`use.names`	Passed through to `base::unlist()`.

Details

For `priority_queue`, this unwraps queue entries to payload items before unlisting (equivalent to `unlist(as.list(x, drop_meta = TRUE), ...)`).

Inherited by `ordered_sequence` and `interval_index` through the shared class stack.

Value

An atomic vector built from `as.list.flexseq()`.

Examples

```
x <- flexseq(1, 2, 3)
unlist(x)

q <- priority_queue("a", "b", priorities = c(2, 1))
unlist(q)
```

upper_bound *Find First Element with Key > Query*

Description

Find First Element with Key > Query

Usage

```
upper_bound(x, key)
```

Arguments

x	An ordered_sequence.
key	Query key.

Details

upper_bound() finds the first element with key > key. This skips exact key matches, which is useful for exclusive range endpoints and for finding the position immediately after a duplicate-key run.

Value

A list with fields:

- found: logical flag.
- index: one-based position of the first match, or NULL.
- value: matched element, or NULL.
- key: matched key, or NULL.

See Also

lower_bound()

Examples

```
x <- ordered_sequence("a", "b", "c", keys = c(1, 2, 2))
upper_bound(x, 2)
upper_bound(x, 10)
```

validate_name_state *Validate name-state invariants only (debug/test utility)*

Description

Checks that trees are either fully unnamed or fully named with unique, non-empty names.

Usage

```
validate_name_state(t)
```

Arguments

t	FingerTree.

Details

Intended for debugging and tests, not hot runtime paths.

Value

TRUE invisibly; errors if name invariants are violated.

Examples

```
x <- as_flexseq(setNames(as.list(letters[1:4]), letters[1:4]))
validate_name_state(x)
```

validate_tree *Validate full tree invariants (debug/test utility)*

Description

Performs expensive full-tree auditing of:

- structural attributes (monoids/measures) consistency
- global name-state invariants

Usage

```
validate_tree(t)
```

Arguments

t	FingerTree.

Details

Intended for debugging and tests, not hot runtime paths.

Value

`TRUE` invisibly; errors if invariant violations are found.

Examples

```
x <- as_flexseq(letters[1:10])
validate_tree(x)
```

`$.flexseq` *Flexseq Indexing*

Description

Index, replace, and extract elements of a `flexseq` by position or name.

Usage

```
## S3 method for class 'flexseq'
x$name

## S3 replacement method for class 'flexseq'
x$name <- value

## S3 method for class 'flexseq'
x[i, ...]

## S3 method for class 'flexseq'
x[[i, ...]]

## S3 replacement method for class 'flexseq'
x[i] <- value

## S3 replacement method for class 'flexseq'
x[[i]] <- value
```

Arguments

`x`	A `flexseq`.
`name`	Element name (for `$` and `$<-`).
`value`	Replacement values; recycled to selected index length.
`i`	Positive integer indices, character element names, or logical mask. For `[[`, a single integer or character name.
`...`	Unused.

Details

Selector behavior:

- Integer indexing (`[`) returns elements in the requested order.
- Character indexing (`[`) returns elements in requested name order; unknown names become `NULL` elements in the output.
- Logical indexing (`[`) is positional and follows logical-mask selection.

Replacement behavior:

- `[<-` is persistent and recycles `value` to the number of selected positions (with standard R recycling warnings where applicable).
- `[[<-` replaces one element; assigning `NULL` removes that element.

Value

For `$`: the matched element.

For `$<-`: updated tree with one named element replaced.

For `[`: a new `flexseq` containing selected elements in query order. For character indexing, missing names are represented as `NULL` elements.

For `[[`: the extracted element (internal name metadata is removed).

For `[<-`: a new `flexseq` with selected elements replaced.

For `[[<-`: a new `flexseq` with one element replaced.

Examples

```
# $ extracts by name
x <- as_flexseq(setNames(as.list(1:3), c("a", "b", "c")))
x$b

# $<- replaces by name
x <- as_flexseq(setNames(as.list(1:3), c("a", "b", "c")))
x$b <- 20
x$b
x <- as_flexseq(letters[1:6])
x

x2 <- x[c(2, 4, 6)]
x2

# named lookups return NULL for missing names
x3 <- as_flexseq(setNames(as.list(letters[1:4]), c("w", "x", "y", "z")))
x4 <- x3[c("y", "missing", "w")]
x4

# logical indexing supports recycling
x[c(TRUE, FALSE)]

# [[ extracts one element
```

```
x <- as_flexseq(letters[1:5])
x[[3]]

x2 <- as_flexseq(setNames(as.list(letters[1:3]), c("a1", "a2", "a3")))
x2[["a2"]]

# [<- replaces selected elements
x <- as_flexseq(1:6)
x

x2 <- x
x2[c(2, 5)] <- list(20, 50)
x2

# character replacement uses element names
x3 <- as_flexseq(setNames(as.list(1:4), c("a", "b", "c", "d")))
x3[c("d", "a")] <- list(40, 10)
x3

# logical replacement supports recycling
x4 <- x
x4[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)] <- list(1)
x4

# [[<- replaces one element
x <- as_flexseq(letters[1:4])
x2 <- x
x2[[2]] <- "ZZ"
x2

x3 <- as_flexseq(setNames(as.list(1:3), c("x", "y", "z")))
x3[["y"]] <- 99
x3

# assigning NULL removes one element
x4 <- as_flexseq(letters[1:4])
x4[[2]] <- NULL
x4
```

Index

[.flexseq *($.flexseq)*, 83
[.interval_index *(sub-.interval_index)*, 76
[.ordered_sequence *(sub-.ordered_sequence)*, 77
[.priority_queue *(sub-.priority_queue)*, 78
[<-.flexseq *($.flexseq)*, 83
[<-.interval_index *(sub-.interval_index)*, 76
[<-.ordered_sequence *(sub-.ordered_sequence)*, 77
[<-.priority_queue *(sub-.priority_queue)*, 78
[[.flexseq *($.flexseq)*, 83
[[.interval_index *(sub-.interval_index)*, 76
[[.ordered_sequence *(sub-.ordered_sequence)*, 77
[[.priority_queue *(sub-.priority_queue)*, 78
[[<-.flexseq *($.flexseq)*, 83
[[<-.interval_index *(sub-.interval_index)*, 76
[[<-.ordered_sequence *(sub-.ordered_sequence)*, 77
[[<-.priority_queue *(sub-.priority_queue)*, 78
$.flexseq, 83
$.interval_index *(sub-.interval_index)*, 76
$.ordered_sequence *(sub-.ordered_sequence)*, 77
$.priority_queue *(sub-.priority_queue)*, 78
$<-.flexseq *($.flexseq)*, 83
$<-.interval_index *(sub-.interval_index)*, 76
$<-.ordered_sequence *(sub-.ordered_sequence)*, 77
$<-.priority_queue *(sub-.priority_queue)*, 78

add_monoids, 4
add_monoids(), *19*, *27*, *32*, *52*, *75*
as.list(), *8*, *10*, *11*, *16*, *20*, *39*, *40*, *42*, *43*, *54*, *56–58*, *80*
as.list.flexseq, 5
as.list.flexseq(), *80*
as.list.interval_index, 6
as.list.ordered_sequence, 6
as.list.priority_queue, 7
as_flexseq, 8
as_flexseq(), *11*, *14*, *18*
as_interval_index, 9
as_iterator.flexseq, 10
as_iterator.priority_queue, 11
as_iterator.priority_queue(), *10*
as_ordered_sequence, 12
as_priority_queue, 13

base::unlist(), *80*

c(), *32*
c.flexseq, 13
coro::loop(), *28*
count_between, 14
count_key, 15

elements_between, 16
elements_between(), *15*

fapply, 17
fapply(), *11*, *20*
flexseq, 18
flexseq(), *8*, *17*

get_measure, 19
get_measure(), *4*, *20*
get_measures, 20

get_measures(), *4*, *19*

insert, 21
insert_at, 22
insert_at(), *21*
interval_index, 23
interval_index(), *8*, *17*, *18*, *21*

length.flexseq, 24
length.interval_index, 24
length.ordered_sequence, 25
length.priority_queue, 26
locate_by_predicate, 26
locate_by_predicate(), *4*
loop, 28
loop(), *10*, *11*
lower_bound, 28
lower_bound(), *81*

max_endpoint, 29
max_key, 30
max_priority, 30
measure_monoid, 31
measure_monoid(), *4*, *19*, *20*, *27*, *52*, *75*
merge.flexseq, 32
merge.interval_index, 33
merge.interval_index(), *33*
merge.ordered_sequence, 34
merge.ordered_sequence(), *33*
merge.priority_queue, 35
merge.priority_queue(), *33*
min_endpoint, 36
min_key, 37
min_priority, 37

ordered_sequence, 38
ordered_sequence(), *8*, *17*, *18*, *21*

peek_all_containing, 39
peek_all_containing(), *45*
peek_all_key, 39
peek_all_max, 40
peek_all_min, 41
peek_all_overlaps, 41
peek_all_overlaps(), *49*
peek_all_point, 42
peek_all_point(), *50*
peek_all_within, 43
peek_all_within(), *50*
peek_at, 44
peek_back, 44
peek_containing, 45
peek_front, 46
peek_key, 46
peek_max, 47
peek_min, 48
peek_min(), *11*
peek_overlaps, 48
peek_point, 49
peek_point(), *42*, *57*, *66*
peek_within, 50
plot_structure, 51
pop_all_containing, 53
pop_all_containing(), *61*
pop_all_key, 54
pop_all_max, 55
pop_all_min, 55
pop_all_overlaps, 56
pop_all_overlaps(), *65*
pop_all_point, 57
pop_all_point(), *66*
pop_all_within, 58
pop_all_within(), *66*
pop_at, 59
pop_back, 60
pop_containing, 61
pop_front, 61
pop_key, 62
pop_max, 63
pop_min, 64
pop_min(), *11*
pop_overlaps, 65
pop_point, 65
pop_within, 66
print.flexseq, 67
print.interval_index, 68
print.ordered_sequence, 69
print.priority_queue, 70
priority_queue, 70
priority_queue(), *8*, *17*, *18*, *21*
push_back, 71
push_front, 72

split_around_by_predicate, 73
split_around_by_predicate(), *4*, *27*, *74*, *75*
split_at, 74
split_by_predicate, 75

split_by_predicate(), *4*, *74*
sub-.interval_index, 76
sub-.ordered_sequence, 77
sub-.priority_queue, 78

unlist(), *20*
unlist.flexseq, 80
upper_bound, 81
upper_bound(), *29*

validate_name_state, 82
validate_tree, 82