Package ‘datastructures’

August 10, 2020

Type Package

Title Implementation of Core Data Structures

Version 0.2.9

Maintainer Simon Dirmeier <simon.dirmeier@web.de>

Description Implementation of advanced data structures such as hashmaps, heaps, or queues. Advanced data structures are essential in many computer science and statistics problems, for example graph algorithms or string analysis. The package uses 'Boost' and 'STL' data types and extends these to R with 'Rcpp' modules.

URL https://github.com/dirmeier/datastructures

BugReports https://github.com/dirmeier/datastructures/issues

License GPL-3

Encoding UTF-8

Depends R (>= 3.3), Rcpp

Suggests testthat, knitr, styler, rmarkdown, lintr

VignetteBuilder knitr

RoxygenNote 6.0.1

SystemRequirements C++11

Imports methods, purrr

LinkingTo Rcpp, BH

NeedsCompilation yes

Collate 'checks.R' 'datastructures-package.R' 'methods_clear.R' 'methods_insert.R' 'methods_size.R' 'methods_pop.R' 'methods_peek.R' 'ds_deque.R' 'ds_deque_queue.R' 'ds_deque_stack.R' 'methods_values.R' 'methods_decrease.R' 'methods_handle.R' 'ds_heap.R' 'ds_heap_binomial.R' 'ds_heap_fibonacci.R' 'methods_erase.R' 'ds_map.R' 'methods_keys.R' 'methods_at.R' 'ds_map_bimap.R' 'ds_map_unordered.R' 'ds_map_hashmap.R' 'ds_map_multimap.R' 'zzz.R'

Author Simon Dirmeier [aut, cre]

Repository CRAN

Date/Publication 2020-08-10 11:20:02 UTC

R topics documented:

`datastructures-package` *datastructures*

Description

Implementation of advanced data structures such as hashmaps, heaps, or queues. Advanced data structures are essential in many computer science and statistics problems, for example graph algorithms or string analysis. The package uses 'Boost' and 'STL' data types and extends these to R with 'Rcpp' modules.

Author(s)

Simon Dirmeier

`at` *Access elements from an object*

Description

Extracts a set of <key, value> pairs. For `hashmaps` mappings from

$$f : keys -> values,$$

exist so argument `which` is per default `values` (since these are going to be retrieved). For `bimaps` also

$$f : values -> keys,$$

mappings exist, such that `which` can also be `keys` if the keys from the object should be retrieved.

Usage

```
at(obj, x, which = c("values", "keys"), ...)

## S4 method for signature 'bimap,vector,character'
at(obj, x, which = c("values", "keys"))

## S4 method for signature 'bimap,vector,missing'
at(obj, x)

## S4 method for signature 'unordered_map,vector,missing'
at(obj, x)
```

Arguments

`obj`	object to extract values from
`x`	the set of keys to match the values
`which`	choose either `values` if the values should get returned
`...`	other arguments or `keys` if the keys should get returned

Details

# datastructures: Implementation of core datastructures for R. # # Copyright (C) Simon Dirmeier # # This file is part of datastructures. # # datastructures is free software: you can redistribute it and/or modify # it under the terms of the GNU General Public License as published by # the Free Software Foundation, either version 3 of the License, or # (at your option) any later version. # # datastructures is distributed in the hope that it will be useful, # but WITHOUT ANY WARRANTY; without even the implied warranty of # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the # GNU General Public License for more details. # # You should have received a copy of the GNU General Public License # along with datastructures. If not, see <http://www.gnu.org/licenses/>.

Value

returns extracted keys or values from `obj`

Examples

```
# access values from a hashmap
h_map <- hashmap("integer")
h_map[seq(2)] <- list(data.frame(a=rexp(3), b=rnorm(3)), environment())
h_map[1L]

# access values or keys from a bimap
b_map <- bimap("integer", "character")
b_map[seq(5)] <- letters[seq(5)]
at(b_map, c(1L, 3L))
at(b_map, c(1L, 3L), which="values")
at(b_map, c("a", "c"), which="keys")

# access values from a multimap
m_map <- multimap("integer")
m_map[c(seq(5), seq(5))] <- letters[seq(10)]
at(m_map, 1L)
```

`bimap` *Create a new* `bimap`

Description

Instantiates a new `bimap` object, i.e. an unordered collection of key-value pairs with mappings

$$f : keys -> values,$$

and

$$f : values -> keys.$$

Usage

```
bimap(key.class = c("character", "numeric", "integer"),
  value.class = c("character", "numeric", "integer"))
```

Arguments

`key.class`	the primitive class type of the keys
`value.class`	the primitive class type of the values

Value

returns a new `bimap` object

Examples

```
# create a bimap with character <-> character bi-mapping
b <- bimap()

# create a bimap with character <-> integer bi-mapping
b <- bimap("character", "integer")

# create a bimap with integer <-> integer bi-mapping
b <- bimap("integer", "numeric")
```

bimap-class *Bimap class*

Description

Implementation of a bimap data structure, i.e. an unordered collection of key-value pairs. The notable difference to `hashmap` is that the mapping is not only

$$f : keys -> values,$$

but also

$$f : values -> keys.$$

Inserting and accessing is amortized in *O(1)*. `bimap` wraps a `boost::bimap` using Rcpp modules.

Slots

`.map` C++ object representing a mapping

`.key.class` the class of the keys

`.value.class` the class of the values

See Also

`bimap` for creating a new `bimap` object

binomial_heap *Create a new* `binomial_heap`

Description

Instantiates a new `binomial_heap` object, i.e. a tree-like data structure satisfying the *min-heap* property.

Usage

```
binomial_heap(key.class = c("character", "numeric", "integer"))
```

Arguments

`key.class`	the primitive class type of the keys

Value

returns a new `binomial_heap` object

Examples

```
# creates a binomial_heap<character, SEXP>
b_heap <- binomial_heap()

# creates a binomial_heap<numeric, SEXP>
b_heap <- binomial_heap("numeric")

# creates a binomial_heap<character, SEXP>
b_heap <- binomial_heap("character")
```

`binomial_heap-class` *Binomial heap class*

Description

Implementation of a binomial heap data structure, i.e. a priority datastructure with push and pop in amortized *O(log n)*. `binomial_heap` wraps a `boost::binomial_heap` using Rcpp modules. The heap consists of nodes with keys and values where the key determines the priority in the heap. Also see the `binomial_heap` class.

Slots

`.heap` C++ object representing a heap

`.key.class` the class of the keys

See Also

`binomial_heap` for creating a new `binomial_heap` object

`clear` *Remove all elements from a datastructure*

Description

Removes every element that is stored in a data structure and resets everything.

Usage

```
clear(obj)

## S4 method for signature 'deque'
clear(obj)

## S4 method for signature 'heap'
```

```
clear(obj)

## S4 method for signature 'map'
clear(obj)
```

Arguments

`obj`	the object to clear

Examples

```
 # clears a multimap
 m_map <- multimap()
 m_map <- insert(m_map, c("a", "b"), 1:2)
 m_map <- insert(m_map, c("a", "b"), list(1, list(a=1)))
 m_map <- clear(m_map)


 # clears a heap
 f_heap <- fibonacci_heap("integer")
 f_heap <- insert(f_heap, 1:2, 1:2)
 f_heap[3:4] <-  list(1, list(a=1))
 f_heap <- clear(f_heap)

 # clears a \code{deque}
 s <- stack()
 s <- insert(s, list(1, vector(), list(3), data.frame(rnorm(3))))
 s <- clear(s)
```

`decrease_key` *Decreases the key of a node in a heap*

Description

Decreases the key of a node in a heap and updates the complete heap. The key is decreases `from` a value `to` a value by that moving the node's position in the heap. If a node cannot uniquely be identified using the `to` key, a `handle` needs to be given in addition.

Usage

```
decrease_key(obj, from, to, handle)

## S4 method for signature 'heap,vector,vector,character'
decrease_key(obj, from, to, handle)

## S4 method for signature 'heap,vector,vector,missing'
decrease_key(obj, from, to)
```

Arguments

`obj`	a heap object
`from`	a key in the heap for which the node should be decreased
`to`	the new value of the heap
`handle`	the handle of the specific node that is decreased

Value

returns extracted handles and values from `obj`

Examples

```
# decreases the key of a heap
f_heap <- fibonacci_heap("integer")
f_heap <- insert(f_heap, 1:5, letters[1:5])
peek(f_heap)

decrease_key(f_heap, 5L, -1L)
peek(f_heap)

hand <- handle(f_heap, value=letters[3])
decrease_key(f_heap, hand[[1]]$key, -2L)
peek(f_heap)
```

`deque-class` *Deque class*

Description

Abstract deque class

Slots

`.deque` `C++` object representing a deque

erase — *Erase an entry from a map*

Description

Erase a vector of key-value pair from a `map` object.

Usage

```
erase(obj, key, value)

## S4 method for signature 'map,vector,missing'
erase(obj, key)

## S4 method for signature 'bimap,missing,vector'
erase(obj, value)

## S4 method for signature 'multimap,vector,vector'
erase(obj, key, value)

## S4 method for signature 'multimap,vector,list'
erase(obj, key, value)

## S4 method for signature 'multimap,vector,ANY'
erase(obj, key, value)
```

Arguments

`obj`	the object to pop an element from
`key`	a vector of keys that should be removed
`value`	optionally a list of values needs to be supplied for some data structures such as `multimaps` if a single key-value pair should removed. If not provided removes all key-value pairs with a specific key.

Value

returns `obj` with removed values

Examples

```
# erases keys from a hashmap or bimap
h_map <- hashmap()
h_map[letters] <- rnorm(length(letters))
h_map <- erase(h_map, "a")
h_map <- erase(h_map, letters[2:5])
```

```
# erases keys from a multimap
m_map <- multimap()
m_map[c("a", "a", "a", "b", "b", "c")] <- rep(1:2, 3)
m_map <- erase(m_map, "a")
m_map <- erase(m_map, "b", 1)
```

fibonacci_heap — *Create a new* `fibonacci_heap`

Description

Instantiates a new `fibonacci_heap` object, i.e. a tree-like data structure satisfying the *min-heap* property.

Usage

```
fibonacci_heap(key.class = c("character", "numeric", "integer"))
```

Arguments

`key.class`	the primitive class type of the keys

Value

returns a new `fibonacci_heap` object

Examples

```
# creates a fibonacci_heap<character, SEXP>
f_heap <- fibonacci_heap()

# creates a fibonacci_heap<numeric, SEXP>
f_heap <- fibonacci_heap("numeric")

# creates a fibonacci_heap<character, SEXP>
f_heap <- fibonacci_heap("character")
```

fibonacci_heap-class *Fibonacci heap class*

Description

Implementation of a Fibonacci heap data structure, i.e. a priority datastructure with push in amortized *O(1)* and pop in *O(log n)*. `fibonacci_heap` wraps a `boost::fibonacci_heap` using Rcpp modules. The heap consists of nodes with keys and values where the key determines the priority in the heap. Also see the `binomial_heap` class.

Slots

`.heap` C++ object representing a heap

`.key.class` the class of the keys

See Also

`fibonacci_heap` for creating a new `fibonacci_heap` object

handle *Get the handles and values for nodes of a specific key in a heap.*

Description

Returns a list of handles and values for node elements that have a specific key. That means for a given key, the reference to the node (the handle) as well as the value of the node are returned. If one key fits fits multiple nodes, all of the values and handles are returned. This is needed in order to uniquely identify a node if, for example, `decrease_key` on a specific node is going to be called.

Usage

```
handle(obj, key, value)

## S4 method for signature 'heap,vector,missing'
handle(obj, key)

## S4 method for signature 'heap,missing,list'
handle(obj, value)

## S4 method for signature 'heap,missing,vector'
handle(obj, value)

## S4 method for signature 'heap,missing,matrix'
handle(obj, value)
```

Arguments

obj	a heap object
key	a key in the heap
value	a value in the heap

Value

returns extracted handles and values from `obj`

Examples

```
# returns the handle of a heap
f_heap <- fibonacci_heap("integer")
f_heap <- insert(f_heap, 1:5, letters[1:5])

handle(f_heap, key=3L)

handle(f_heap, value=letters[3])
```

`hashmap` *Create a new* `hashmap`

Description

Instantiates a new `hashmap` object, i.e. an unordered collection of key-value pairs with mapping

$$f : keys- > values$$

, where only unique key-value pairs can be stored.

Usage

```
hashmap(key.class = c("character", "numeric", "integer"))
```

Arguments

key.class	the primitive class type of the keys

Value

returns a new hashmap object

Examples

```
# creates a hashmap<character, SEXP>
h <- hashmap()

# creates a hashmap<integer, SEXP>
h <- hashmap("integer")

# creates a hashmap<numeric, SEXP>
h <- hashmap("numeric")
```

`hashmap-class` *Hashmap class*

Description

Implementation of a hashmap data structure, i.e. an unordered collection of key-value pairs:

$$f : keys -> values.$$

Hashmaps only to store unique keys-value pairs. For a data structure where multiple identical keys can be stores see `multimap`. Inserting and accessing is amortized in *O(1)*. `hashmap` wraps a C++ `unordered_map` using Rcpp modules. Also see `bimap` for mappings in both ways.

Slots

`.map` C++ object representing a mapping

`.key.class` the class of the keys

See Also

`hashmap` for creating a new `hashmap` object

`heap-class` *Abstract heap class*

Description

Abstract heap class

Slots

`.heap` C++ object representing a heap

`.key.class` the class of the keys

`insert` *Add elements to an object*

Description

Adds keys or <key, value> pairs to an object and returns the object. Depending on the datastructure used, either only keys are required or pairs of <keys, values>. Insertion of elements with vectors, i.e. giving multiple arguments at the same time is faster than inserting elements iteratively.

Usage

```
insert(obj, x, y)

## S4 method for signature 'deque,ANY,missing'
insert(obj, x)

## S4 method for signature 'deque,list,missing'
insert(obj, x)

## S4 method for signature 'heap,vector,vector'
insert(obj, x, y)

## S4 method for signature 'heap,vector,matrix'
insert(obj, x, y)

## S4 method for signature 'heap,vector,list'
insert(obj, x, y)

## S4 method for signature 'heap,vector,ANY'
insert(obj, x, y)

## S4 method for signature 'bimap,vector,vector'
insert(obj, x, y)

## S4 method for signature 'unordered_map,vector,vector'
insert(obj, x, y)

## S4 method for signature 'unordered_map,vector,list'
insert(obj, x, y)

## S4 method for signature 'unordered_map,vector,ANY'
insert(obj, x, y)
```

Arguments

`obj`	object to insert into
`x`	the values/keys to insert into

y	values to be inserted which are required for some datastructures

Value

returns obj with inserted values

Examples

```
# inserts values into a multimap
m_map <- multimap()
m_map <- insert(m_map, c("a", "b"), 1:2)
m_map <- insert(m_map, c("a", "b"), list(1, list(a=1)))
m_map["a"] <- rnorm(length(letters))
m_map[c("a", "b", "c")] <- list(1, data.frame(a=2), environment())

# inserts values into a fibonacci_heap
f_heap <- fibonacci_heap("integer")
f_heap <- insert(f_heap, 1:2, 1:2)
f_heap[3:4] <- list(1, list(a=1))
f_heap <- insert(f_heap, 5:6, list(data.frame(a=rnorm(3)), diag(2)))

# inserts elements into a queue or stack
s <- stack()
s <- insert(s, list(1, vector(), list(3), data.frame(rnorm(3))))
```

---

keys *Get keys from an object*

Description

Extracts the keys from a map object.

Usage

```
keys(obj)

## S4 method for signature 'bimap'
keys(obj)

## S4 method for signature 'unordered_map'
keys(obj)
```

Arguments

obj	object to extract keys from

Value

returns the extracted keys as vector

Examples

```
# returns the keys of a hashmap
h_map <- hashmap("numeric")
h_map[rnorm(3)] <- list(1, 2, 3)
keys(h_map)

# returns the keys of a multimap
m_map <- multimap("numeric")
m_map[c(1, 2, 1)] <- list(rnorm(1), rgamma(1, 1), rexp(1))
keys(m_map)
```

`map-class` *Map class*

Description

Abstract map class

Slots

`.map` C++ object representing a mapping

`.key.class` the class of the keys

`multimap` *Create a new* `multimap`

Description

Instantiates a new `multimap` object, i.e. an unordered collection of key-value pairs with mapping

$$f : keys -> values$$

, where multiple identical key-value paors can be stored.

Usage

```
multimap(key.class = c("character", "numeric", "integer"))
```

Arguments

`key.class`	the primitive class type of the keys

Value

returns a new `multimap` object

Examples

```
# creates a new multimap<character, SEXP>
m <- multimap()

# creates a new multimap<numeric, SEXP>
m <- multimap("numeric")

# creates a new multimap<character, SEXP>
m <- multimap("integer")
```

`multimap-class` *Multimap class*

Description

Implementation of a multimap data structure, i.e. an unordered collection of key-value pairs:

$$f : keys -> values.$$

Multimaps are able to store several identical keys. For a data structure which unique keys, see `hashmap`. Inserting and accessing is amortized in *O(1)*. `hashmap` wraps a C++ `unordered_multimap` using Rcpp modules. Also see `bimap` for mappings in both ways.

Slots

`.map` C++ object representing a mapping

`.key.class` the class of the keys

See Also

`multimap` for creating a new `multimap` object

peek — *Have a look at the first element from an object without removing it*

Description

Peeks into an object, i.e. takes the first element and returns it without removing it from the object. The data structure that has a peek method usually uses some sort of priority of its elements.

Usage

```
peek(obj)

## S4 method for signature 'deque'
peek(obj)

## S4 method for signature 'heap'
peek(obj)

## S4 method for signature 'map'
peek(obj)
```

Arguments

`obj`	the object to peek

Value

returns the first element from `obj` as list

Examples

```
# peeks into a queue
q <- queue()
q <- insert(q, list(environment(), data.frame(a=1)))
peek(q)

# peeks into a fibonacci heap
b_heap <- binomial_heap()
b_heap <- insert(b_heap, letters[seq(3)], list(1, diag(3), rnorm(2)))
peek(b_heap)

# peeks into a \code{hashmap}
h_map <- hashmap()
h_map[letters] <- rnorm(length(letters))
peek(h_map)

# peeks into a \code{bimap}
b_map <- bimap("integer", "integer")
```

```
b_map[seq(10)] <- seq(10, 1)
peek(b_map)
```

---

pop *Pop a single element from an object*

Description

Remove and return the first element from a data structure that has a priority, such as a `heap` or `deque`.

Usage

```
pop(obj)

## S4 method for signature 'deque'
pop(obj)

## S4 method for signature 'heap'
pop(obj)
```

Arguments

`obj`	the object to pop an element from

Value

returns the first element from `obj` as list

Examples

```

# pops from a queue
q <- queue()
q <- insert(q, list(environment(), data.frame(a=1)))
pop(q)

# pops from a stack
s <- stack()
s <- insert(s, list(environment(), data.frame(a=1)))
pop(s)

# pops from a fibonacci heap
b_heap <- binomial_heap()
b_heap <- insert(b_heap, letters[seq(3)], list(1, diag(3), rnorm(2)))
pop(b_heap)
```

`queue` *Create a new* `queue`

Description

Instantiates a new `queue` object, i.e. a list implementation with FIFO principle.

Usage

```
queue()
```

Value

returns a new queue object

Examples

```
# returns a new queue<SEXP>
q <- queue()
```

`queue-class` *Queue class*

Description

Implementation of a queue data structure, i.e. a list implementation with FIFO principle. `queue` uses a `std::deque` as default container, so inserting, peeking and popping functions require constant *O(1)*. See `stack` for a class using the LIFO principle.

Slots

`.deque` `C++` object representing a deque

See Also

`queue` for creating a new queue object.

size *Get the size of an object*

Description

Computes the size of an object, i.e. the number of keys or <key, value> pairs depending on the object.

Usage

```
size(obj)

## S4 method for signature 'deque'
size(obj)

## S4 method for signature 'heap'
size(obj)

## S4 method for signature 'map'
size(obj)
```

Arguments

`obj`	the object to get the size from

Value

returns the size of `obj`

Examples

```
# get the size of a hashmap
h_map <- hashmap()
h_map[letters] <- rnorm(length(letters))
size(h_map)

# get the size of a fibonacci heap
f_heap <- fibonacci_heap()
f_heap <- insert(f_heap, letters[seq(3)], list(1, diag(3), rnorm(2)))
size(f_heap)

# get the size of a stack
s <- stack()
s <- insert(s, list(1))
size(s)
```

`stack` *Create a new* `stack`

Description

Instantiates a new `stack` object, i.e. a list implementation with LIFO principle.

Usage

```
stack(...)
```

Arguments

`...`	parameters that are only needed if `utils::stack` should be called

Value

returns a new `stack` object

Examples

```
# creates a new stack<SEXP>
s <- stack()
```

`stack-class` *Stack class*

Description

Implementation of a stack data structure, i.e. a list implementation with LIFO principle. `stack` uses a `std::deque` as default container, so inserting, peeking and popping functions require constant *O(1)*. See `queue` for a class using the FIFO principle.

Slots

`.deque` C++ object representing a deque

See Also

`stack` for creating a new `stack` object.

`unordered_map-class` *Abstract unordered map class*

Description

Abstract unordered map class

Slots

`.map` C++ object representing a mapping

`.key.class` the class of the keys

`values` *Get values from an object*

Description

Extracts the values from a data structure such as a `map` or `heap` object.

Usage

```
values(obj)

## S4 method for signature 'heap'
values(obj)

## S4 method for signature 'bimap'
values(obj)

## S4 method for signature 'unordered_map'
values(obj)
```

Arguments

`obj`	object to extract values from

Value

returns the extracted values as a `list` or, when primitive, as a `vector`. In case of a `heap` also returns `key` and `handle` of the heap node.

Examples

```
# shows the values of a hashmap
h_map <- hashmap("integer")
h_map <- insert(h_map, seq(2), list(data.frame(a=1), 3))
values(h_map)

# shows the values of a multimap
m_map <- multimap("integer")
m_map[seq(2)] <- list(diag(2), rnorm(3))
values(m_map)

# shows the values of a heap
f_heap <- fibonacci_heap("integer")
f_heap <- insert(f_heap, 1:2, list(diag(2), rnorm(3)))
values(f_heap)
```

`[,unordered_map,vector,missing,missing-method`
Extract elements from an object

Description

Access <key, value> pairs of an unordered map using a set of keys.

Usage

```
## S4 method for signature 'unordered_map,vector,missing,missing'
x[i]
```

Arguments

`x`	an unorderd map object, such as a `hashmap` or `multimap`
`i`	a vector of keys

`[<-,bimap,vector,missing,vector-method`
Insert parts to an object

Description

Inserts <key, value> pairs to a bimap.

Usage

```
## S4 replacement method for signature 'bimap,vector,missing,vector'
x[i] <- value
```

Arguments

`x`	a `map` object
`i`	a vector of keys
`value`	a vector of values for the keys

`[<-,heap,vector,missing,list-method`
Insert parts to an object

Description

Inserts <key, value> pairs to a heap. The keys are determine the ordering of the heap, while the value is the actual value to store.

Usage

```
## S4 replacement method for signature 'heap,vector,missing,list'
x[i] <- value
```

Arguments

`x`	a heap object, such as a `fibonacci_heap` or a `binomial_heap`
`i`	a vector of keys
`value`	a vector of values for the keys

`[<-,heap,vector,missing,matrix-method`
Insert parts to an object

Description

Inserts <key, value> pairs to a heap. The keys are determine the ordering of the heap, while the value is the actual value to store.

Usage

```
## S4 replacement method for signature 'heap,vector,missing,matrix'
x[i] <- value
```

Arguments

`x`	a heap object, such as a `fibonacci_heap` or a `binomial_heap`
`i`	a vector of keys
`value`	a vector of values for the keys

`[<-,heap,vector,missing,vector-method`
Insert parts to an object

Description

Inserts <key, value> pairs to a heap. The keys are determine the ordering of the heap, while the value is the actual value to store.

Usage

```
## S4 replacement method for signature 'heap,vector,missing,vector'
x[i] <- value
```

Arguments

`x`	a heap object, such as a `fibonacci_heap` or a `binomial_heap`
`i`	a vector of keys
`value`	a vector of values for the keys

`[<-,unordered_map,vector,missing,ANY-method`
Insert parts to an object

Description

Inserts <key, value> pairs to an unordered_map.

Usage

```
## S4 replacement method for signature 'unordered_map,vector,missing,ANY'
x[i] <- value
```

Arguments

`x`	x an unorderd map object, such as a `hashmap` or `multimap`
`i`	a vector of keys
`value`	a vector of values for the keys

[<-,unordered_map,vector,missing,list-method
Insert parts to an object

Description

Inserts <key, value> pairs to an unordered_map.

Usage

```
## S4 replacement method for signature 'unordered_map,vector,missing,list'
x[i] <- value
```

Arguments

x	x an unorderd map object, such as a hashmap or multimap
i	a vector of keys
value	a vector of values for the keys

[<-,unordered_map,vector,missing,vector-method
Insert parts to an object

Description

Inserts <key, value> pairs to an unordered_map.

Usage

```
## S4 replacement method for signature 'unordered_map,vector,missing,vector'
x[i] <- value
```

Arguments

x	x an unorderd map object, such as a hashmap or multimap
i	a vector of keys
value	a vector of values for the keys

Index